达梦数据库管理系统

DM SQL 语言使用手册

武汉华工达梦数据库有限公司

2005 年 09 月

目 录

第 1 章　结构化查询语言 DM_SQL 简介

结构化查询语言 SQL(Structured Query Language)是在 1974 年提出的一种关系数据库语言。由于 SQL 语言接近英语的语句结构，方便简洁、使用灵活、功能强大，倍受用户及计算机工业界的欢迎，被众多计算机公司和数据库厂商所采用，经各公司的不断修改、扩充和完善，SQL 语言最终发展成为关系数据库的标准语言。

SQL 的第一个标准是 1986 年 10 月由美国国家标准化组织(ANSI)公布的 ANSI X3.135-1986 数据库语言 SQL，简称 SQL-86，1987 年国际标准化组织(ISO)也通过了这一标准。以后通过对 SQL-86 的不断修改和完善，于 1989 年第二次公布了 SQL 标准 ISO/IEC 9075-1989(E)，即 SQL-89。1992 年又公布了 SQL 标准 ISO/IEC 9075：1992，即 SQL-92。最新的 SQL 标准是 SQL-3（也称 SQL-99），1999 年作为 ISO/IEC 9075：1999《信息技术——数据库语言 SQL》发布。我国也相继公布了数据库语言 SQL 的国家标准。

在 SQL 成为国际标准以后，其影响远远超出了数据库领域，例如，在 CAD、软件工程、人工智能、分布式等领域，不仅把 SQL 作为检索数据的语言规范，而且也把 SQL 作为检索图形、图象、声音、文字、知识等信息类型的语言规范。目前，世界上大型的著名数据库管理系统均支持 SQL 语言，如 Oracle、Sybase、SQL Server、DB2 等。在未来相当长的时间里，SQL 仍将是数据库领域以至信息领域中数据处理的主流语言之一。

由于不同的 DBMS 产品，大都按自己产品的特点对 SQL 语言进行了扩充，很难完全符合 SQL 标准。目前在 DBMS 市场上已将 SQL 的符合率作为衡量产品质量的重要指标，并研制成专门的测试软件，如 NIST。目前， DM SQL-92 入门级和过渡级的符合率均达到 100%，并且部分支持更新的 SQL-99 标准。同时 DM 还兼容 Oracle 8i 和 SQL Server 2000 的部分语言特性。本章主要介绍 DM 系统所支持的 SQL 语言——DM_SQL 语言。

1.1　DM_SQL 语言的特点

DM_SQL 语言符合结构化查询语言 SQL 标准，是标准 SQL 的扩充。它集数据定义、数据查询、数据操纵和数据控制于一体，是一种统一的、综合的关系数据库语言。它功能强大，使用简单方便、容易为用户掌握。DM_SQL 语言具有如下特点：

1．功能一体化

DM_SQL 的功能一体化表现在以下两个方面：

(1) DM_SQL 支持多媒体数据类型，用户在建表时可直接使用。DM 系统在处理常规数据与多媒体数据时达到了四个一体化：一体化定义、一体化存储、一体化检索、一体化处理，最大限度地提高了数据库管理系统处理多媒体的能力和速度。

(2) DM_SQL 语言集数据库的定义、查询、更新、控制、维护、恢复、安全等一系列操作于一体，每一操作都只需一种操作符表示，格式规范，风格一致，简单方便，很容易为用户所掌握。

2．两种用户接口使用统一语法结构的语言

DM_SQL 语言既是自含式语言，又是嵌入式语言。作为自含式语言，它能独立运行于

联机交互方式。作为嵌入式语言，DM_SQL 语句能够嵌入到 C 和 C++语言程序中，将高级语言(也称主语言)灵活的表达能力、强大的计算功能与 DM_SQL 语言的数据处理功能相结合，完成各种复杂的事务处理。而在这两种不同的使用方式中，DM_SQL 语言的语法结构是一致的，从而为用户使用提供了极大的方便性和灵活性。

3．高度非过程化

DM_SQL 语言是一种非过程化语言。用户只需指出“做什么”，而不需指出“怎么做”，对数据存取路径的选择以及 DM_SQL 语句功能的实现均由系统自动完成，使用户编制的应用程序与具体的机器及关系 DBMS 的实现细节无关，从而方便了用户，提高了应用程序的开发效率，也增强了数据独立性和应用系统的可移植性。

4．面向集合的操作方式

DM_SQL 语言采用了集合操作方式。不仅查询结果可以是元组的集合，而且一次插入、删除、修改操作的对象也可以是元组的集合，相对于面向记录的数据库语言(一次只能操作一条记录)来说，DM_SQL 语言的使用简化了用户的处理，提高了应用程序的运行效率。

5．语言简洁，方便易学

DM_SQL 语言功能强大，格式规范，表达简洁，接近英语的语法结构，容易为用户所掌握。

1.2　保留字与标识符

标识符的语法规则兼容标准 GJB 1382A-9X，标识符分为正规标识符和定界标识符两大类。

正规标识符以字母、_、$、#或汉字开头，后面可以跟随字母、数字、_、$、#或者汉字，正规标识符的最大长度是 128 个英文字符或 64 个汉字。正规标识符不能是保留字。

正规标识符的例子：A，test1，_TABLE_B，表 1。

定界标识符的标识符体用双引号括起来时，标识符体可以包含任意字符，特别地，其中使用连续两个双引号转义为一个双引号。

定界标识符的例子："table"，"A"，"!@#$"。

保留字的清单参见附录 1。

1.3　字符集

不同国家和地区往往使用不同的字符集。仅中文就有 GB2312、GBK 等简体字符集和台湾、香港等不同的繁体字符集。此外，英文、日文、韩文等字符集也互不相同，且不一定能互相转化。一个优秀的软件应能适应不同国家和地区使用不同的字符集的现实需要。

Unicode 标准为全球商业领域中广泛使用的大部分字符定义了一个单一编码方案。Unicode 数据中的位模式被一致地翻译成字符，保证了同一个位模式在所有的计算机上总是转换成同一个字符。Unicode 数据可以随意地从一个数据库或计算机传送到另一个数据库或计算机，而不用担心接收系统是否会错误地翻译位模式。一般各种字符集，都

能与 Unicode 相互转换。所以，支持 Unicode 的系统往往可支持多种字符集。

DM 系统内部支持 Unicode 字符集(具体地说，是 UTF-8 字符集)，具有支持各种字符集的扩展能力。DM 目前支持以下多个字符集（对其它字符集的支持须定制）：

```
UTF-8          /*压缩 Unicode */
GBK            /*中国大陆，汉语 */
BIG5           /*中国台湾地区，汉语 */
ISO-8859-9     /* Latin-5，土耳其语 */
EUC_JP         /*日文版 Unix */
EUC_KR         /*韩语版 Unix */
KOI8           /* 俄罗斯*/
```

安装 DM 时,用户可以选择在 DM 服务器端是使用操作系统缺省字符集还是使用 UTF-8 字符集来存储数据。

选择前者时，用户须保证客户端、服务器端使用相同的缺省字符集。这时 DM 不考虑字符集转换因素，直接按缺省字符集处理，效率较高。

选择后者时，用户可以随时使用系统提供的接口（JDBC、API）设置客户端使用的字符集（缺省使用客户端操作系统的缺省字符集）。设置之后，用户就可以透明地使用该字符集。DM 负责不同字符集之间的自动转换，但有一定的时间、空间消耗。

1.4　DM_SQL 语言的功能及语句

DM_SQL 语言是一种介于关系代数与关系演算之间的语言，其功能主要包括数据定义、查询、操纵和控制四个方面，通过各种不同的 SQL 语句来实现。按照所实现的功能，DM_SQL 语句分为以下几种：

1．数据库、登录、用户、模式、基表、视图、索引、序列、全文索引、存储过程和触发器的定义和删除语句，登录、基表、视图、全文索引的修改语句，对象的更名语句；

2．查询（含全文检索）、插入、删除、修改语句；

3．数据库安全语句。包括创建角色、删除角色语句，授权语句、回收权限语句，修改登录口令语句，审计设置语句、取消审计设置语句等。

在嵌入方式中，为了协调 DM_SQL 语言与主语言不同的数据处理方式，DM_SQL 语言引入了游标的概念。因此在嵌入方式下，除了数据查询语句(一次查询一条记录)外，还有几种与游标有关的语句：

1．游标的定义、打开、关闭、拨动语句；

2．游标定位方式的数据修改与删除语句。

为了有效维护数据库的完整性和一致性，支持 DBMS 的并发控制机制，DM_SQL 语言提供了事务的回滚（ROLLBACK）与提交（COMMIT）语句。同时 DM 允许选择实施事务级读一致性，它保证同一事务内的可重复读，为此 DM 提供用户多种手动上锁语句，和设置事务隔离级别语句。

1.5　DM_SQL 所支持的数据类型

数据类型是可表示值的集。值的逻辑表示是<字值>。值的物理表示依赖于实现。DM 系统具有 SQL-92 的绝大部分数据类型，以及部分 SQL-99 和 SQL Server 2000 的数据类型。

1.5.1　常规数据类型

1．字符数据类型

CHAR 类型

语法：CHAR[(长度)]

功能：CHAR 数据类型指定定长字符串。在基表中，定义 CHAR 类型的列时，可以指定一个不超过 8188 的正整数作为字符长度，例如：CHAR(100)。如果未指定长度，缺省为 1。DM 确保存储在该列的所有值都具有这一长度。CHAR 数据类型的最大长度由数据库页面大小决定，字符类型最大长度和页面大小的对应关系请见下表 5.1.1。DM 支持按字节存放字符串。

表 5.1.1

数据库页面大小	最大长度
4K	1900
8K	3900
16K	8000
32K	8188

CHARACTER 类型

语法：CHARACTER[(长度)]

功能：与 CHAR 相同。

VARCHAR 类型

语法：VARCHAR[(长度)]

功能：VARCHAR 数据类型指定变长字符串，用法类似 CHAR 数据类型，可以指定一个不超过 8188 的正整数作为字符长度，例如：VARCHAR (100)。如果未指定长度，缺省为 1。在 DM 系统中，VARCHAR 数据类型的实际最大长度由数据库页面大小决定，具体最大长度算法如表 5.1.2。CHAR 同 VARCHAR 的区别在于前者长度不足时，系统自动填充空格，而后者只占用实际的字节空间。

表 5.1.2

数据库页面大小	实际最大长度
4K	1900
8K	3900

16K	8000
32K	8188

2．数值数据类型

(1) 精确数值数据类型

NUMERIC 类型

语法：NUMERIC[(精度 [, 标度])]

功能：NUMERIC 数据类型用于存储零、正负定点数。其中：精度是一个无符号整数，定义了总的数字数，精度范围是 1 至 38，标度定义了小数点右边的数字位数，定义时如省略精度，则默认是 16。如省略标度，则默认是 0。一个数的标度不应大于其精度。例如：NUMERIC(4,1)定义了小数点前面 3 位和小数点后面 1 位，共 4 位的数字，范围在-999.9 到 999.9。所有 NUMERIC 数据类型，如果其值超过精度，DM 返回一个出错信息，如果超过标度，则多余的位截断。

DECIMAL 类型

语法：DECIMAL[(精度 [, 标度])]

功能：与 NUMERIC 相似。

DEC 类型

语法：DEC[(精度[, 标度])]

功能：与 DECIMAL 相同。

MONEY 类型

语法：MONEY

功能：该类型用于存储货币数据，货币数据表示正的或负的货币值。货币数据存储的精度为 19，标度为 4 的精确数值。货币数值用句点将局部的货币单位（如角分）从总体货币单位中分隔出来。例如 2.15 表示 2 元 15 分。

BIT 类型

语法：BIT

功能：BIT 类型用于存储整数数据 1、0 或 NULL，可以用来支持 ODBC 和 JDBC 的布尔数据类型。DM 的 BIT 类型与 SQL SERVER2000 的 BIT 数据类型相似。

INTEGER 类型

语法：INTEGER

功能： 用于存储有符号整数，精度为 10，标度为 0。取值范围为：-2147483648 (-2^31)～ +2147483647(2^31-1)。

INT 类型

语法：INT

功能：与 INTEGER 相同。

BIGINT 类型

语法：BIGINT

功能：用于存储有符号整数,精度为 19，标度为 0。取值范围为：-9223372036854775808(-2^63)～9223372036854775807(2^63-1)。

TINYINT 类型

语法：TINYINT

功能：用于存储有符号整数，精度为 3，标度为 0。取值范围为：-128 ～ +127。

BYTE 类型

语法：BYTE

功能：与 TINYINT 相似，精度为 3，标度为 0。

SMALLINT 类型

语法：SMALLINT

功能：用于存储有符号整数，精度为 5，标度为 0。取值范围为：-32768(-2^15 ～ +32767(2^15-1)。

BINARY 类型

语法：BINARY[(长度)]

功能：BINARY 数据类型指定定长二进制数据。缺省长度为 1 个字节。 最大长度由数据库页面大小决定，具体最大长度算法与 CHAR 类型的相同。BINARY 常量以 0x 开始，后面跟着数据的十六进制表示。例如 0x2A3B4058。

VARBINARY 类型

语法：VARBINARY[(长度)]

功能：VARBINARY 数据类型指定变长二进制数据，用法类似 BINARY 数据类型，可以指定一个不超过 8188 的正整数作为数据长度。 缺省长度为 1 个字节。VARBINARY 数据类型的实际最大长度由数据库页面大小决定，具体最大长度算法与 VARCHAR 类型的相同。

(2) 近似数值数据类型

FLOAT 类型

语法：FLOAT[(精度)]

功能：FLOAT 是带二进制精度的浮点数，精度最大不超过 53，如省略精度，则二进制精度为 53，十进制精度为 15。取值范围为-1.7E + 308 ～ 1.7E + 308。

DOUBLE 类型

语法：DOUBLE[(精度)]

功能：同 FLOAT 相似，精度最大不超过 53。

REAL 类型

语法：REAL

功能：REAL 是带二进制的浮点数，但它不能由用户指定使用的精度，系统指定其二进制精度为 24，十进制精度为 7。取值范围-3.4E + 38 ～ 3.4E + 38。

DOUBLE PRECISION 类型

语法：DOUBLE PRECISION

功能：该类型指明双精度浮点数，其二进制精度为 53，十进制精度为 15。取值范围 -1.7E + 308 ～ 1.7E + 308。

1.5.2　日期时间数据类型

日期时间数据类型分为一般日期时间数据类型和时间间隔数据类型两类，用于存储日期、时间和它们之间的间隔信息。

1．一般日期时间数据类型

DATE 类型

语法：DATE

功能：DATE 类型包括年、月、日信息，定义了'0001-01-01'和'9999-12-31'之间任何一个有效的格里高利日期。

DM 支持 SQL92 标准或 SQL SERVER 的 DATE 字值，例如：DATE '1999-10-01'、'1999-10-01'、'1999/10/01'或'1999.10.01'都是有效的 DATE 值，且彼此等价。年月日中第一个非 0 数值前的 0 亦可省略，例如'0001-01-01'等价于'1-1-1'。

TIME 类型

语法：TIME[(小数秒精度)]

功能：TIME 类型包括时、分、秒信息，定义了一个在'00:00:00.000000'和'23:59:59.999999'之间的有效时间。TIME 类型的小数秒精度规定了秒字段中小数点后面的位数，取值范围为 0～6，如果未定义，缺省精度为 6。

DM 支持 SQL92 标准或 SQL SERVER 的 TIME 字值，例如：TIME'09:10:21', '09:10:21'或'9:10:21' 都是有效的 TIME 值，且彼此等价。

TIMESTAMP 类型

语法：TIMESTAMP[(小数秒精度)]

功能：TIMESTAMP 类型包括年、月、日、时、分、秒信息，定义了一个在'0001-01-01 00:00:00.000000'和'9999-12-31 23:59:59.999999'之间的有效格里高利日期时间。TIMESTAMP 类型的小数秒精度规定了秒字段中小数点后面的位数，取值范围为 0～

6，如果未定义，缺省精度为 6。

DM 支持 SQL92 标准或 SQL SERVER 的 TIMESTAMP 字值，例如：TIMESTAMP ‘2002-12-12 09:10:21’或‘2002-12-12 9:10:21’或‘2002/12/12 09:10:21’或‘2002.12.12 09:10:21’ 都是有效的 TIMESTAMP 值，且彼此等价。

语法中，TIMESTAMP 也可以写为 DATETIME。

2．时间间隔数据类型

DM 支持两类十三种时间间隔类型：两类是年-月间隔类和日-时间隔类，它们通过时间间隔限定符区分，前者结合了日期字段年和月，后者结合了时间字段日、时、分、秒。由时间间隔数据类型所描述的值总是有符号的。

(1) 年-月间隔类

INTERVAL YEAR TO MONTH 类型

语法：INTERVAL YEAR［(引导精度)］TO MONTH

功能：描述一个若干年若干月的间隔，引导精度规定了年的取值范围，而月的取值范围在 0 到 11 之间，如果不定义引导精度默认精度为 2。例如：INTERVAL YEAR(4) TO MONTH，其中 YEAR(4)表示年的精度为 4，表示范围为负 9999 年零 12 月到正 9999 年零 12 月。一个合适的字值例子是：INTERVAL ‘0015-08’ YEAR TO MONTH。

INTERVAL YEAR 类型

语法：INTERVAL YEAR［(引导精度)］

功能：描述一个若干年的间隔，引导精度规定了年的取值范围，如果不定义引导精度默认精度为 2。例如：INTERVAL YEAR(4)，其中 YEAR(4)表示年的精度为 4，表示范围为负 9999 年到正 9999 年。一个合适的字值例子是：INTERVAL ‘0015’ YEAR。

INTERVAL MONTH 类型

语法：INTERVAL MONTH［(引导精度)］

功能：描述一个若干月的间隔，引导精度规定了月的取值范围，如果不定义引导精度默认精度为 2。例如：INTERVAL MONTH(4)，其中 MONTH(4)表示月的精度为 4，表示范围为负 9999 月到正 9999 月。一个合适的字值例子是：INTERVAL ‘0015’ MONTH。

(2) 日-时间隔类

INTERVAL DAY 类型

语法：INTERVAL DAY［(引导精度)］

功能：描述一个若干日的间隔，引导精度规定了日的取值范围，如果不定义引导精度默认精度为 2。例如：INTERVAL DAY(3)，其中 DAY (3)表示日的精度为 3，表示范围为负 999 日到正 999 日。一个合适的字值例子是：INTERVAL ‘150’ DAY。

INTERVAL DAY TO HOUR 类型

语法：INTERVAL DAY［(引导精度)］ TO HOUR

功能：描述一个若干日若干小时的间隔，引导精度规定了日的取值范围，如果不定义引导精度默认精度为 2。而时的取值范围在 0 到 23 之间。例如：INTERVAL DAY(1) TO HOUR，其中 DAY(1)表示日的精度为 1，表示范围为负 9 日零 23 小时到正 9 日零 23 小时。一个合适的字值例子是：INTERVAL '9 23' DAY TO HOUR。

INTERVAL DAY TO MINUTE 类型

语法：INTERVAL DAY［(引导精度)］ TO MINUTE

功能：描述一个若干日若干小时若干分钟的间隔，引导精度规定了日的取值范围，如果不定义引导精度默认精度为 2。而小时的取值范围在 0 到 23 之间，分钟的取值范围在 0 到 59 之间。例如：INTERVAL DAY(2) TO MINUTE，其中 DAY(2)表示日的精度为 2，表示范围为负 99 日零 23 小时零 59 分到正 99 日零 23 小时零 59 分。一个合适的字值例子是：INTERVAL '09 23:12' DAY TO MINUTE。

INTERVAL DAY TO SECOND 类型

语法：INTERVAL DAY［(引导精度)］ TO SECOND [(小数秒精度)]

功能：描述一个若干日若干小时若干分钟若干秒的间隔，引导精度规定了日的取值范围，如果不定义引导精度默认精度为 2，小数秒精度规定了秒字段中小数点后面的位数，取值范围为 0～6, 如果不定义小数秒精度默认精度为 6。小时的取值范围在 0 到 23 之间，分钟的取值范围在 0 到 59 之间。例如：INTERVAL DAY(2) TO SECOND(1)，其中 DAY(2)表示日的精度为 2，SECOND(1)表示秒的小数点后面取 1 位，表示范围为负 99 日零 23 小时零 59 分零 59.9 秒到正 99 日零 23 小时零 59 分零 59.9 秒。一个合适的字值例子是：INTERVAL '09 23:12:01.1' DAY TO SECOND。

INTERVAL HOUR 类型

语法：INTERVAL HOUR［(引导精度)］

功能：描述一个若干小时的间隔，引导精度规定了小时的取值范围，如果不定义引导精度默认精度为 2。例如：INTERVAL HOUR(3)，其中 HOUR (3)表示时的精度为 3，表示范围为负 999 小时到正 999 小时。一个合适的字值例子是：INTERVAL '150' HOUR。

INTERVAL HOUR TO MINUTE 类型

语法：INTERVAL HOUR［(引导精度)］ TO MINUTE

功能：描述一个若干小时若干分钟的间隔，引导精度规定了小时的取值范围，如果不定义引导精度默认精度为 2。而分钟的取值范围在 0 到 59 之间。例如：INTERVAL HOUR(2) TO MINUTE，其中 HOUR(2)表示小时的精度为 2，表示范围为负 99 小时零 59 分到正 99 小时零 59 分。一个合适的字值例子是：INTERVAL '23:12' HOUR TO MINUTE。

INTERVAL HOUR TO SECOND 类型

语法：INTERVAL HOUR［（引导精度）］ TO SECOND [(小数秒精度)]

功能：描述一个若干小时若干分钟若干秒的间隔，引导精度规定了小时的取值范围，如果不定义引导精度默认精度为 2，小数秒精度规定了秒字段中小数点后面的位数，取值范围为 0～6，如果不定义小数秒精度默认精度为 6。分钟的取值范围在 0 到 59 之间。例如：INTERVAL HOUR(2) TO SECOND(1)，其中 HOUR(2)表示小时的精度为 2，SECOND(1)表示秒的小数点后面取 1 位，表示范围为负 99 小时零 59 分零 59.9 秒到正 99 小时零 59 分零 59.9 秒。一个合适的字值例子是：INTERVAL '23:12:01.1' HOUR TO SECOND。

INTERVAL MINUTE 类型

语法：INTERVAL MINUTE［（引导精度）］

功能：描述一个若干分钟的间隔，引导精度规定了分钟的取值范围，如果不定义引导精度默认精度为 2。例如：INTERVAL MINUTE(3)，其中 MINUTE(3)表示分钟的精度为 3，表示范围为负 999 分钟到正 999 分钟。一个合适的字值例子是：INTERVAL '150' MINUTE。

INTERVAL MINUTE TO SECOND 类型

语法：INTERVAL MINUTE［（引导精度）］ TO SECOND [(小数秒精度)]

功能：描述一个若干分钟若干秒的间隔，引导精度规定了分钟的取值范围，如果不定义引导精度默认精度为 2，小数秒精度规定了秒字段中小数点后面的位数，取值范围为 0～6，如果不定义小数秒精度默认精度为 6。例如：INTERVAL MINUTE(2) TO SECOND(1)，其中 MINUTE(2)表示分钟的精度为 2，SECOND(1)表示秒的小数点后面取 1 位，表示范围为负 99 分零 59.9 秒到正 99 分零 59.9 秒。一个合适的字值例子是：INTERVAL '12:01.1' MINUTE TO SECOND。

INTERVAL SECOND 类型

语法：INTERVAL SECOND［（引导精度 [,小数秒精度])]

功能：描述一个若干秒的间隔，引导精度规定了秒整数部分的取值范围，如果不定义引导精度默认精度为 2，小数秒精度规定了秒字段中小数点后面的位数，取值范围为 0～6，如果不定义小数秒精度默认精度为 6。例如：INTERVAL SECOND(2,1)，表示范围为负 99.9 秒到正 99.9 秒。一个合适的字值例子是：INTERVAL '51.1' SECOND。

1.5.3　多媒体数据类型

TEXT 类型

语法：TEXT

功能：TEXT 为变长字符串类型。其字符串的长度最大为 2G-1。DM 利用它存储长的文本串。

LONGVARCHAR（又名 TEXT）类型

语法：LONGVARCHAR

功能：与 TEXT 相同。

IMAGE 类型

语法：IMAGE

功能：IMAGE 用于指明多媒体信息中的图像类型。图像由不定长的象素点阵组成，长度最大为 2G-1 字节。该类型除了存储图像数据之外，还可用于存储任何其它二进制数据。

LONGVARBINARY（又名 IMAGE）类型

语法：LONGVARBINARY

功能：与 IMAGE 相同。

BLOB 类型

语法：BLOB[(长度)]

功能：BLOB 类型用于指明一个长度从 1 到定义长度的变长二进制大对象。其中长度是一个无符号正整数，长度以字节为单位。BLOB 的长度最大为 2G-1 字节。定义长度指明了在 BLOB 字段中可接受的最大字节长度数值，例如 BLOB(1000)，则该字段长度不得超过 1000 字节。若不定义长度，缺省长度为 2G-1。

CLOB 类型

语法：CLOB[(长度)]

功能：CLOB 类型用于指明一个长度从 1 到定义长度的变长字母数字字符串。其中长度是一个无符号正整数，长度以字节为单位。CLOB 的长度最大为 2G-1 字节。定义长度指明了在 CLOB 字段中可接受的最大字节长度数值，例如 CLOB(1000) ，则该字段长度不得超过 1000 字节。若不定义长度，缺省长度为 2G-1。

注：多媒体数据类型的字值有两种格式，一种类似于字符串类型，例如：‘ABCD’，一种类似于 BINARY，如 0x61626364。

1.6　DM_SQL 语言支持的表达式

DM 支持多种类型的表达式，包括数值表达式、字符串表达式、时间值表达式、时间间隔值表达式等。本节中引用的数据库实例请参见第 2 章。

1.6.1　数值表达式

1．一元算符 + 和 -

语法：+exp 、-exp

(exp 代表表达式，下同)

当单独使用时，+ 和 - 改变表达式的符号。

例：

```
SELECT -(-5), +资产总值
FROM 厂商登记
WHERE 资产总值<50;
```

结果是：

```
5        30.00
5        20.00
```

注意：在 SQL 中由于两短横即 --表示“注释开始”，则双负号必须是-(-5)，而不是--5。

2．一元算符 ~

语法：~exp

它是按位非算符，要求参与运算的操作数都为整数数据类型。

例：

```
SELECT ~10
FROM 厂商登记
WHERE 资产总值<50;
```

结果是：

```
-11
-11
```

3．二元算符 +、-、*、/

语法：exp1+exp2 、exp1- exp2 、exp1*exp2 、exp1/exp2

(exp1 代表表达式 1，exp2 代表表达式 2，下同)

当在表达式之间使用+、-、*、/ 时，分别表示加、减、乘、除运算。对于结果的精度规定如下：

(1) 只有精确数值数据类型的运算

两个相同类型的整数运算的结果类型不变，两个不同类型的整数运算的结果类型转换为范围较大的那个整数类型。

整数与 NUMERIC，DEC 等类型运算时，SMALLINT 类型的精度固定为 5，标度为 0；INTEGER 类型的精度固定为 10，标度为 0；BIGINT 类型的精度固定为 19，标度为 0。

exp1+exp2 结果的精度为二者整数部分长度(即精度－标度)的最大值与二者标度的

最大值之和，标度是二者标度的最大值；

exp1-exp2 结果的精度为二者整数部分长度的最大值与二者标度的最大值之和，标度是二者标度的最大值；

exp1*exp2 结果的精度为二者的精度之和，标度也是二者的标度之和；

exp1/exp2 结果的标度为下面两个值中的较大者：

值一：exp1 标度 + exp2 精度 - exp2 标度 + 1。

值二：exp2 标度 + 1。

exp1/exp2 结果的精度为：上面求得的标度 + exp1 精度 - exp1 标度 + exp2 标度。

例：

```
SELECT 资产总值+10, 资产总值-10
FROM 厂商登记
WHERE 资产总值<50;
```

结果是：

40.00　20.00
30.00　10.00

例：

```
SELECT 资产总值*10, 资产总值/10
FROM 厂商登记
WHERE 资产总值<50;
```

结果是：

300.00　3.0000000000000
200.00　2.0000000000000

(2) 有近似数值数据类型的运算

对于 exp1+exp2 、exp1- exp2 、exp1*exp2 、exp1/exp2 中 exp1 和 exp2 只要有一个为近似数值数据类型，则结果为近似数值数据类型。

例：

```
CREATE TABLE APPROXIMATE(
        C1 FLOAT(53),
        C2 REAL,
        C3 DOUBLE PRECISION
);
INSERT INTO APPROXIMATE VALUES(5E+20,5E+2,5E+2);
SELECT C1+50,C2-50,C2*C3,C3/50 FROM APPROXIMATE;
```

结果是：

500000000000000000000.0000000000　　450.0000000000
250000.0000000000　　10.0000000000

4．二元算符 &

语法：exp1 & exp2

它是按位与算符，要求参与运算的操作数都为整数数据类型。

例：

```
SELECT 20 & 10 FROM 厂商登记 WHERE 资产总值<50;
```

结果是：

0
0

5．二元算符 |

语法：exp1 | exp2

它是按位或算符，要求参与运算的操作数都为整数数据类型。

例：

```
SELECT 20 | 10 FROM 厂商登记 WHERE 资产总值<50;
```

结果是：

30
30

6．二元算符 ^

语法：exp1 ^ exp2

它是按位异或算符，要求参与运算的操作数都为整数数据类型。

例：

```
SELECT 20 ^ 10 FROM 厂商登记 WHERE 资产总值<50;
```

结果是：

30
30

1.6.2　字符串表达式

连接 ||

语法：STR1 || STR2

(STR1 代表字符串 1，STR2 代表字符串 2)

连接操作符对两个运算数进行运算，其中每一个都是对属于同一字符集的字符串的求值。它以给定的顺序将字符串连接在一起，并返回一个字符串。其长度等于两个运算数长度之和。如果两个运算数中有一个是 NULL，则运算的结果是 NULL。

例：

```
SELECT '中国' || 厂商名 FROM 厂商登记;
```

结果是：

中国华乐电视机厂

中国日东冰箱厂
中国万里鞋厂
中国美的服装厂
中国海天电视机厂
中国小鸭洗衣机厂

1.6.3　时间值表达式

时间值表达式的结果为时间值类型，包括日期(DATE)类型，时间(TIME)类型和时间戳(TIMESTAMP)间隔类型。DM SQL 不是对于任何的日期时间和间隔运算数的组合都可以计算。如果任何一个运算数是 NULL，运算结果也是 NULL。下面列出了有效的可能性和结果的数据类型。

1．日期+间隔，日期-间隔和间隔+日期，得到日期

日期表达式的计算是根据有效格里高利历日期的规则。如果结果是一个无效的日期，表达式将出错。参与运算的间隔类型只能是 INTERVAL YEAR、INTERVAL MONTH、INTERVAL YEAR TO MONTH、INTERVAL DAY。

如果间隔运算数是年-月间隔，则没有从运算数的 DAY 字段的进位。

例：由第 2 章可知，供货登记表中采购商品数量为 10 的进货日期为'1998-10-31'。

```
SELECT 进货日期+INTERVAL '1' MONTH
FROM 供货登记 WHERE 采购商品数量=10;
```

结果是：

服务器报错。因为'＋'的结果 DATE '1998-11-31'是一个无效的日期。

例：

```
SELECT 进货日期+INTERVAL '1' YEAR, 进货日期-INTERVAL '1' YEAR
FROM 供货登记 WHERE 采购商品数量<50;
```

结果是：

```
1999-10-05    1997-10-05
1999-10-31    1997-10-31
```

2．时间+间隔，时间-间隔和间隔+时间，得到时间

时间表达式的计算是根据有效格里高利历日期的规则。如果结果是一个无效的时间，表达式将出错。参与运算的间隔类型只能是 INTERVAL DAY、INTERVAL HOUR、INTERVAL MINUTE、INTERVAL SECOND、INTERVAL DAY TO HOUR、INTERVAL DAY TO MINUTE、INTERVAL SECOND、INTERVAL HOUR TO MINUTE、INTERVAL HOUR TO SECOND、INTERVAL MINUTE TO SECOND。

当结果的小时值大于等于 24 时，时间表达式是对 24 模的计算。

例：

```
SELECT TIME '19:00:00'+INTERVAL '9' HOUR,
       TIME '19:00:00'-INTERVAL '9' HOUR
FROM 供货登记 WHERE 采购商品数量=10;
```

结果是：

04:00:00　　　　10:00:00

3．时间戳记+间隔，时间戳记-间隔和间隔+时间戳记，得到时间戳记

时间戳记表达式的计算是根据有效格里高利历日期的规则。如果结果是一个无效的时间戳记，表达式将出错。参与运算的间隔类型只能是 INTERVAL YEAR、INTERVAL MONTH、INTERVAL YEAR TO MONTH、INTERVAL DAY、INTERVAL DAY、INTERVAL HOUR、INTERVAL MINUTE、INTERVAL SECOND、INTERVAL DAY TO HOUR、INTERVAL DAY TO MINUTE、INTERVAL SECOND、INTERVAL HOUR TO MINUTE、INTERVAL HOUR TO SECOND、INTERVAL MINUTE TO SECOND。

与时间的计算不同，当结果的小时值大于等于 24 时，结果进位到天。

例：

```
SELECT TIMESTAMP'2003-07-15 19:00:00'+INTERVAL'9'HOUR,
       TIMESTAMP'2003-07-15 19:00:00'-INTERVAL'9'HOUR
FROM 供货登记 WHERE 采购商品数量=10;
```

结果是：

2003-07-16 04:00:00.0　　　2003-07-15 10:00:00.0

注意：在含有 SECOND 值的运算数之间的一个运算的结果具有等于运算数的小数秒精度的小数秒精度。

4．日期+数值，日期-数值和数值+日期，得到日期

日期与数值的运算，等价于日期与一个 INTERVAL ‘数值’ DAY 的时间间隔的运算。

例：

```
SELECT CURDATE();
```

结果是：

2003-09-29 /* 假设该查询操作发生在 2003 年 9 月 29 日 */

```
SELECT CURDATE() + 2;
```

结果是：

2003-10-01

```
SELECT CURDATE() - 100;
```

结果是：

2003-06-21

5．时间戳记+数值，时间戳记-数值和数值+时间戳记，得到时间戳记

时间戳记与数值的运算，将数值看作以天为单位，转化为一个 INTERVAL DAY TO SECOND(6)的时间间隔，然后进行运算。

例：时间戳加上 2.358 天。

```
SELECT TIMESTAMP '2003-09-29 08:59:59.123' + 2.358;
```

结果是：

2003-10-01 17:35:30.323

1.6.4　时间间隔值表达式

1．日期-日期，得到间隔

由于得到的结果可能会是“年-月-日”间隔，而这是不支持的间隔类型，故要对结果强制使用语法：

（日期表达式-日期表达式）<时间间隔限定符>

结果由<时间间隔限定符>中最不重要的日期字段决定。

例：

```
SELECT (进货日期-DATE'1990-01-01')YEAR TO MONTH
FROM 供货登记 WHERE 采购商品数量<50;
```

结果是：

```
INTERVAL '8-9' YEAR TO MONTH
INTERVAL '8-9' YEAR TO MONTH
```

2．时间 － 时间，得到间隔

要对结果强制使用语法：

（时间表达式-时间表达式）<时间间隔限定符>

结果由<时间间隔限定符>中最不重要的时间字段决定。

例：

```
SELECT (TIME'19:00:00'-TIME'10:00:00') HOUR
FROM 供货登记 WHERE 采购商品数量=10;
```

结果是：

```
INTERVAL '9' HOUR
```

3．时间戳记 - 时间戳记，得到间隔

要对结果强制使用语法：

（时间戳记表达式-时间戳记表达式）<时间间隔限定符>

结果由<时间间隔限定符>中最不重要的日期时间字段决定。

例：

```
SELECT (TIMESTAMP '2003-07-15 19:00:00'- TIMESTAMP '2003-01-15 19:00:00') HOUR
FROM 供货登记 WHERE 采购商品数量=10;
```

结果是：

```
INTERVAL '4344' HOUR
```

4．年月间隔 + 年月间隔和年月间隔 - 年月间隔，得到年月间隔

参加运算的两个间隔必须有相同的数据类型，若得出无效的间隔的表达式将出错。结果的子类型包括运算数子类型所有的域，关于结果的引导精度规定如下：

如果二者的子类型相同，则为二者引导精度的最大值；如果二者的子类型不同，则

为与结果类型首字段相同的那个运算数的引导精度。

例：

```
SELECT INTERVAL'2003-07'YEAR(4) TO MONTH + INTERVAL'7'MONTH,
       INTERVAL'2003-07'YEAR(4) TO MONTH - INTERVAL'7'MONTH
FROM 供货登记 WHERE 采购商品数量=10;
```

结果是：

INTERVAL '2003-14' YEAR TO MONTH　　INTERVAL '2003-0' YEAR TO MONTH

5．日时间隔 + 日时间隔和日时间隔 - 日时间隔，得到日时间隔

参加运算的两个间隔必须有相同的数据类型，若得出无效的间隔的表达式将出错。结果的子类型包含运算数子类型所有的域，结果的小数秒精度为两运算数的小数秒精度的最大值，关于结果的引导精度规定如下：

如果二者的子类型相同，则为二者引导精度的最大值；如果二者的子类型不同，则为与结果类型首字段相同的那个运算数的引导精度。

例：

```
SELECT INTERVAL'7 15'DAY TO HOUR + INTERVAL'10:10'MINUTE TO SECOND,
       INTERVAL'7 15'DAY TO HOUR - INTERVAL'10:10'MINUTE TO SECOND
FROM 供货登记 WHERE 采购商品数量=10;
```

结果是：

INTERVAL '7 15:10:10.0' DAY TO SECOND

INTERVAL '7 15:-10:-10.0' DAY TO SECOND

6．间隔 * 数字，间隔 / 数字和数字 * 间隔，得到间隔

若得出无效的间隔的表达式将出错。结果的子类型、小数秒精度、引导精度和原间隔相同。

例：

```
SELECT INTERVAL'10:10'MINUTE TO SECOND * 3,
       INTERVAL'10:10'MINUTE TO SECOND / 3
FROM 供货登记 WHERE 采购商品数量=10;
```

结果是：

INTERVAL '30:30.0' MINUTE TO SECOND

INTERVAL '3:23.333333' MINUTE TO SECOND

1.6.5　运算符的优先级

当一个复杂的表达式有多个运算符时，运算符优先性决定执行运算的先后次序。运算符有下面这些优先等级（从高到低排列）。在较低等级的运算符之前先对较高等级的

运算符进行求值。

```
()
+（一元正）、-（一元负）、~（一元按位非）
*（乘）、/（除）
+（加）、|| （串联）、-（减）
^（位异或）、&（位与）、|（位或）
```

1.7 DM_SQL 语言支持的数据库模式

DM_SQL 语言支持关系数据库的三级模式，外模式对应于视图和部分基表，模式对应于基表，基表是独立存在的表。一个或若干个基表存放于一个存贮文件中，存贮文件中的逻辑结构组成了关系数据库的内模式。DM_SQL 语言本身不提供对内模式的操纵语句。

视图是从基表或其它视图上导出的表，DM 只将视图的定义保存在数据字典中。该定义实际为一查询语句，再为该查询语句取一名字即为视图名。每次调用该视图时，实际上是执行其对应的查询语句，导出的查询结果即为该视图的数据。所以视图并无自己的数据，它是一个虚表，其数据仍存放在导出该视图的基表之中。当基表中的数据改变时，视图中查询的数据也随之改变，因此，视图象一个窗口，用户透过它可看到自己权限内的数据。视图一旦定义也可以为多个用户所共享，对视图作类似于基表的一些操作就象对基表一样方便。

综上所述，SQL 语言对关系数据库三级模式的支持如图 1.7.1 所示。

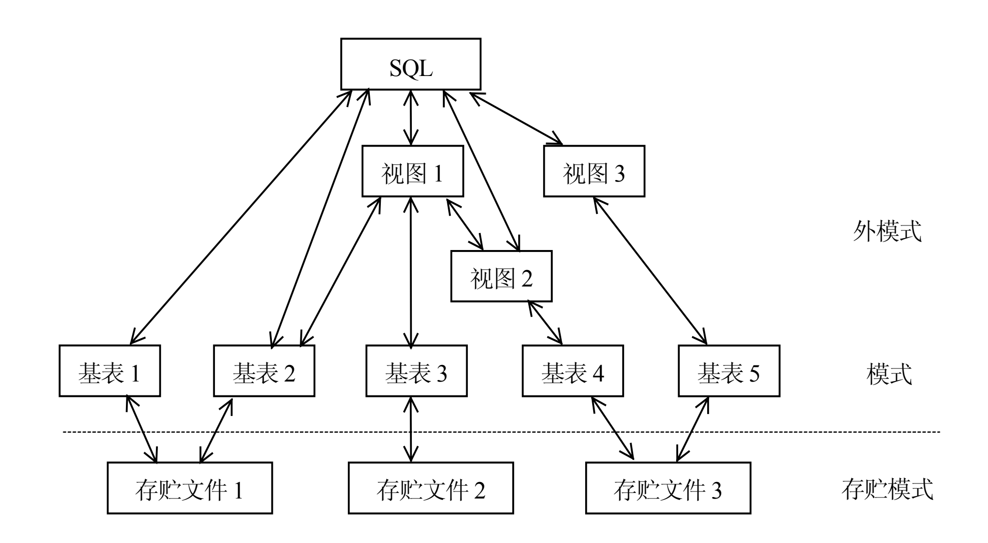


图 1.7.1　关系数据库的三级模式

第 2 章　本手册中的实例说明

为方便读者阅读并尽快学会使用 DM 系统，本手册在介绍利用 DM 建立、维护数据库以及对数据库进行的各种操作中，大部分使用了同一个实例。本章将对该实例进行说明。

2.1　实例说明

某商品采购供应公司主要业务是向各厂商采购商品，供应给各个商场。现欲建立一个商品管理信息系统，数据库中包含有四个基表，表名分别为：厂商登记、商品登记、商场登记、供货登记。各表结构如下：

1．厂商登记　关键字为：厂商编号

列名	列类型	列长度	是否允许为空值
厂商编号	CHAR	5	NOT
厂商名	CHAR	20	
资产总值	NUMERIC(10, 2)		
厂商地址	CHAR	10	

2．商品登记　关键字为：商品编号

列名	列类型	列长度	是否允许为空值
商品编号	CHAR	8	NOT
商品名	CHAR	10	
价格	NUMERIC(10, 2)		
商品照片	IMAGE		
仓库地址	CHAR	10	
厂商编号	CHAR	5	

3．商场登记　关键字为：商场编号

列名	列类型	列长度	是否允许为空值
商场编号	CHAR	5	NOT
商场名	CHAR	20	
商场地址	CHAR	10	

4．供货登记

列名	列类型	列长度	是否允许为空值
商场编号	CHAR	5	NOT
商品编号	CHAR	8	NOT

采购商品数量	SMALLINT		
进货日期	DATE		

说明：

(1) 设厂商不设立其它的分厂，商场不设立其它的连锁店，因此，每一厂商或商场的编号均是唯一的；

(2) 表中的厂商地址、仓库地址、商场地址只限制在北京、上海、武汉三个城市且只登记城市名；

(3) 一种商品只放在一个城市的仓库中；

(4) 在供货登记表中，必须是已登记的厂商将已入库的商品供应给已登记的商场；

(5) 供货登记表只登记各商场从 1998 年 1 月 1 日开始的进货；

(6) 资产总值的单位为万元，价格的单位为元，采购商品数量的单位为人们生活中习惯的用法，如：台、件、个、双等，但不单独存储；

(7) 供货登记表中，以商品编号为索引列建立索引 S1，以商场编号、商品编号、进货日期为索引列建立唯一索引 S2；

(8) 生产家电类商品的厂商编号中的第三个字符为“A”。

2.2　实例中的数据

在以上四个基表中，装有以下数据：

1．厂商登记

厂商编号	厂商名	资产总值	厂商地址
B0A01	华乐电视机厂	50	北京
B0A02	日东冰箱厂	80	武汉
B0003	万里鞋厂	30	北京
B0A04	海天电视机厂	60	武汉
B0A05	小鸭洗衣机厂	1000	上海
B0006	美的服装厂	20	武汉

2．商品登记

商品编号	商品名	价格	商品照片	仓库地址	厂商编号
C0001	电视机	3000		武汉	B0A01
C0002	电冰箱	2000		武汉	B0A02
C0003	鞋_1	50		武汉	B0003
C0004	电视机	2500		上海	B0A04
C0005	洗衣机	1000		上海	B0A05
C0006	鞋_2	500		武汉	B0003
C0007	电视机	2800		北京	B0A04

C0008	电冰箱	1900		上海	B0A02

3．商场登记

商场编号	商 场 名	商场地址
A0001	红旗商场	武汉
A0002	上海商场	上海
A0003	王府井商场	北京
A0004	人民商场	武汉
A0005	首都商城	北京

4．供货登记

商场编号	商品编号	采购商品数量	进货日期
A0001	C0002	100	1998-10-05
A0001	C0004	50	1998-10-18
A0002	C0003	1000	1998-10-20
A0002	C0002	30	1998-10-05
A0003	C0002	10	1998-10-31
A0004	C0005	200	1998-10-20
A0002	C0002	100	1998-10-30

2.3　参考脚本

```
--创建示例数据库（假设数据库页面大小为 8K）
CREATE DATABASE cw DATAFILE 'cw.dbf' SIZE 10;

--创建登录 user1（密码 user1）并设置其默认数据库 cw
CREATE LOGIN user1 IDENTIFIED BY user1;
ALTER LOGIN user1 DEFAULT DATABASE cw;

--在数据库 cw 中创建用户 user1，并将用户 user1 添加到登录 user1 中授予
CREATE USER user1 AT cw;
ALTER LOGIN user1 ADD USER user1 AT cw;

--将 RESOURCE 权限授予给用户 user1;
GRANT RESOURCE TO cw.user1;

--以下脚本请以登录名 user1 登录到 cw 数据库再执行

--创建基表
```

```
CREATE   TABLE   厂商登记
( 厂商编号     CHAR(5)   NOT   NULL,
  厂商名       CHAR(20),
  资产总值     NUMERIC(10,2)   DEFAULT 10.00,
  厂商地址     CHAR(10),
  PRIMARY   KEY(厂商编号),
  CHECK (厂商地址  IN ('北京','上海','武汉'))
);

CREATE   TABLE   商品登记
(  商品编号     CHAR(8)   NOT NULL,
  商品名       CHAR(10),
  价格         NUMERIC(10,2),
  商品照片     IMAGE,
  仓库地址     CHAR(10),
  厂商编号     CHAR(5),
  FOREIGN     KEY(厂商编号)   REFERENCES   厂商登记(厂商编号),
  PRIMARY     KEY(商品编号),
  CHECK(仓库地址  IN ('北京','上海','武汉'))
);

CREATE TABLE   商场登记
(  商场编号       CHAR(5)   NOT NULL,
  商场名           CHAR(20),
  商场地址         CHAR(10),
  PRIMARY           KEY(商场编号),
  CHECK(商场地址    IN    ('北京','上海','武汉'))
);

CREATE TABLE   供货登记
(  商场编号       CHAR(5)   NOT NULL,
  商品编号         CHAR(8)   NOT NULL,
  采购商品数量  SMALLINT,
  进货日期         DATE DEFAULT '1998-01-01',
  FOREIGN           KEY(商场编号)   REFERENCES   商场登记(商场编号),
  FOREIGN           KEY(商品编号)   REFERENCES   商品登记(商品编号)
);

--初始化数据
INSERT INTO  厂商登记
```

```
VALUES ('B0A01','华乐电视机厂',50,'北京');

INSERT INTO 厂商登记
VALUES('B0A02','日东冰箱厂',80,'武汉');

INSERT INTO 厂商登记
VALUES('B0003','万里鞋厂',30,'北京');

INSERT INTO 厂商登记
VALUES('B0A04','海天电视机厂',60,'武汉');

INSERT INTO 厂商登记
VALUES('B0A05','小鸭洗衣机厂',1000,'上海');

INSERT INTO 厂商登记
VALUES('B0006','美的服装厂',20,'武汉');

INSERT INTO 商品登记
VALUES('C0001','电视机',3000,NULL,'武汉','B0A01');

INSERT INTO 商品登记
VALUES('C0002','电冰箱',2000,NULL,'武汉','B0A02');

INSERT INTO 商品登记
VALUES('C0003','鞋_1',50,NULL,'武汉','B0003');

INSERT INTO 商品登记
VALUES('C0004','电视机',2500,NULL,'上海','B0A04');

INSERT INTO 商品登记
VALUES('C0005','洗衣机',1000,NULL,'上海','B0A05');

INSERT INTO 商品登记
VALUES('C0006','鞋_2',500,NULL,'武汉','B0003');

INSERT INTO 商品登记
VALUES('C0007','电视机',2800,NULL,'北京','B0A04');

INSERT INTO 商品登记
VALUES('C0008','电冰箱',1900,NULL,'上海','B0A02');
```

```
INSERT INTO 商场登记
VALUES('A0001','红旗商场','武汉');

INSERT INTO 商场登记
VALUES('A0002','上海商场','上海');

INSERT INTO 商场登记
VALUES('A0003','王府井商场','北京');

INSERT INTO 商场登记
VALUES('A0004','人民商场','武汉');

INSERT INTO 商场登记
VALUES('A0005','首都商场','北京');

INSERT INTO 供货登记
VALUES('A0001','C0002',100,'1998-10-05');

INSERT INTO 供货登记
VALUES('A0001','C0004',50,'1998-10-18');

INSERT INTO 供货登记
VALUES('A0002','C0003',1000,'1998-10-20');

INSERT INTO 供货登记
VALUES('A0002','C0002',30,'1998-10-05');

INSERT INTO 供货登记
VALUES('A0003','C0002',10,'1998-10-31');

INSERT INTO 供货登记
VALUES('A0004','C0005',200,'1998-10-20');

INSERT INTO 供货登记
VALUES('A0002','C0002',100,'1998-10-30');
```

第 3 章　数据定义语句

3.1　数据库定义语句

数据是按一定的数据结构存储在数据库中。从用户的角度看，在 DM 数据库中，数据被组织到用户可以看见的逻辑组件（表、视图）中。从物理实现方式看，DM 数据库是以多个物理文件组成的。

用户主要使用数据库中的逻辑组件，例如表、视图、存储过程和用户。数据库的物理实现在很大程度上对用户是透明的。

DM 有两种由系统创建和管理的数据库：一种是系统数据库（SYSTEM.DBF），用于存放数据字典中的系统表信息以及用户在它上面创建的各种数据对象；另一种是系统运行时创建的临时数据库(TMPDBxxx.DBF)，临时数据库可有多个，具体个数在 DM.INI 文件中进行设置，也就是对 DM.INI 文件中的 TEMP_DBS 项进行赋值。

除上述三种由系统创建和管理的数据库之外，用户可以自行建立一个或多个用户数据库。各个数据库间具有一定的相互独立性，库中所有对象的作用范围都不能超出其所在的库。本节介绍的是用户数据库定义语句。

语句格式

```
CREATE DATABASE <数据库名> DATAFILE ‘<文件路径>’ SIZE <文件大小>;
```

参数

1．<数据库名>　新数据库的名称。数据库名称在服务器中必须唯一，数据库名仅能包含字母、数字、’_’以及非 ASCII 字符，且不能以数字开头。数据库名最多可以包含 128 个字符, 如果没有双引号，数据库名在建立后一律转为大写,如果用户加上双引号则大小写不变。

2．<文件路径>　指明被操作的数据文件在操作系统下的路径＋新数据文件名。数据文件的存放路径符合 DM 安装路径的规则，且该路径必须是已经存在的。

3．<文件大小>　整数值，指明新增数据文件的大小（单位 MB）。

语句功能

供 DBA 或具有 CREATE_DATABASE 权限的用户创建一个新数据库及存储该数据库的文件。

使用说明

1．所指定的数据库和文件不能是已经存在的。

2．文件大小至少 10MB，且不小于 512*每页大小。文件大小的上限由操作系统限定。

3．库文件的页大小、簇大小等在安装时统一指定，以后不能更改。

举例说明

例 创建数据库 MYDB，数据文件存放路径为 C:\DATAFILE(该路径必须存在，否则创建数据库语句不能执行成功)，数据库文件名为 MYDB.DBF，大小为 40Mbyte (假设每

页大小为 8K)。

```
CREATE DATABASE MYDB DATAFILE 'C:\DATAFILE\MYDB.DBF' SIZE 40;
```

3.2　数据库修改语句

一个数据库创建成功后，可以通过增大原有数据库文件大小，或在该数据库中加入新的数据库文件两种方式来修改（增大）该数据库空间。

语句格式

```
ALTER DATABASE <数据库名>
    ADD DATAFILE ‘<文件路径>’ SIZE ‘<文件大小>’ [ON FILEGROUP ‘<    文件组名>’];|
    ADD FILEGROUP ‘<文件组名>’  DEFAULT DATAFILE ‘<文件路径>’ SIZE ‘<文件大小>’; |
        MODIFY DATAFILE ‘<文件路径>’ INCREASE <文件增量>;
```

参数

1．<数据库名>　指明被操作的数据库。

2．<文件路径>　指明被操作的数据文件在操作系统下的路径＋数据文件名。

3．<文件大小>　整数值，指明新增数据文件的大小。

4．<文件增量>　整数值，指明对数据文件的增量大小。

5．<文件组名>　指明要增加的文件组。

语句功能

供具有 DBA 权限的用户修改（增加）数据库空间或增加文件组以及增加文件组中数据文件。

使用说明

1．对于<ADD 子句>，所指定的数据文件在指定文件路径下并不存在，在此命令执行成功后才会生成；对于<MODIFY 子句>，必须对已存在的数据文件进行操作。

2．对于所指定的整数值，其单位是 MB。

举例说明

假设数据库 MYDB 页面大小为 8K，数据文件存放路径为 C:\DATAFILE。

例 1 给 MYDB 数据库增加一个新数据文件 TMP1.DBF，其大小为 40M：

```
ALTER DATABASE MYDB ADD DATAFILE 'C:\DATAFILE\TMP1.DBF' SIZE 40;
```

例 2 扩展 MYDB 数据库中的这个数据文件，使其大小增大为 60M：

```
ALTER DATABASE MYDB MODIFY DATAFILE 'C:\DATAFILE\TMP1.DBF' INCREASE 20;
```

例 3 增加文件组 FG_ORDER 在默认位置：

```
ALTER DATABASE TEST ADD FILEGROUP FG_ORDER DEFAULT DATAFILE 'C:\DMDBMS\DMData\ORDER.DBF' size 50;
```

例 4 增加文件组 FG_TOOL 在其它位置：

```
ALTER DATABASE TEST ADD FILEGROUP FG_TOOL DEFAULT DATAFILE 'C:\EXP\TOOL.DBF' size 50;
```

例 5 在文件组 FG_ORDER 中增加数据文件在默认位置：

```
ALTER DATABASE TEST ADD datafile 'C:\DMDBMS\DMData\ORDER_ADD1.DBF' size 50 on FILEGROUP FG_ORDER;
```

例 6 在文件组 FG_ORDER 中增加数据文件在其它位置：

```
ALTER DATABASE TEST ADD datafile 'C:\EXP\ORDER_ADD2.DBF' size 50 on FILEGROUP FG_ORDER;
```

3.3　数据库删除语句

删除数据库空间。

语句格式

```
DROP DATABASE <数据库名>;
```

参数

<数据库名>　指定要删除的数据库名称。

语句功能

供 DBA 权限的用户删除数据库。

使用说明

1．执行此操作，将删除该数据库及其包含的所有文件。

2．使用者应拥有 DBA 权限。

3．无法删除系统数据库（SYSTEM.DBF、临时数据库）。

举例说明

例 删除用户数据库 MYDB：

```
DROP DATABASE MYDB;
```

3.4　设置当前数据库语句

设置当前活动数据库。

语句格式

```
SET CURRENT DATABASE <数据库名>;
```

参数

<数据库名>　用户指定的当前活动数据库。

语句功能

供用户指定当前活动数据库。

使用说明

如果用户登录系统时指定了要登录到哪个数据库，登录成功后，该数据库就是当前数据库；如果登录时没有指定登录哪个数据库，则登录的将是用户所使用的登录对应的缺省数据库，登录成功后，该缺省数据库就是当前数据库。

该语句供用户指定当前活动数据库，接下来的未指定数据库名的操作都在设定的当前数据库上进行，直至指定另一数据库为当前数据库为止。

举例说明

例 设当前活动数据库为 SYSTEM，用户要在另一个数据库 MYDB 上进行一系列数据操作，则可利用以下语句将当前活动数据库设置为 MYDB。

```
SET CURRENT DATABASE MYDB;
```

3.5　用户定义语句

在数据库中创建新用户。

语句格式

```
CREATE USER <用户名> [AT <数据库名>];
```

参数

1．<用户名>　　指明要创建的用户名。

2．<数据库名>　指明用户所属的数据库，缺省为当前数据库。

语句功能

供具有 CREATE USER 权限的用户在指定数据库创建指定的新用户。

使用说明

1．用户名在一个数据库中必须唯一，且不能与角色名相同。

2．每个数据库都有两个系统自动创建的用户 SYSDBA 和 SYSAUDITOR。

举例说明

例 1．在数据库 MYDB 中创建用户名为 RU 的用户。

```
CREATE USER RU AT MYDB;
```

3.6　用户删除语句

删除数据库中的用户。

语句格式

```
DROP USER   [<数据库名>.]<用户名>   [CASCADE];
```

参数

<用户名>　指明被删除的用户的名字。

<数据库名>指明被删除的用户所属的数据库的名字，缺省为当前数据库。

语句功能

供具有 DBA 权限的用户删除用户。

使用说明

1．只有具有 DBA 权限的用户才能删除用户。

2．执行此语句将导致 DM 删除数据库中用户建立的所有对象，且不可恢复。如果要保存这些实体，请参考 REVOKE 语句。

3．如果未使用 CASCADE 选项，若该用户建立了数据库对象（如表、视图、过程

或函数），或其他用户对象引用了该用户的对象，或在该用户的表上存在其它用户建立的视图，DM 将返回错误信息，而不删除此用户。

4．如果使用了 CASCADE 选项，除数据库中该用户及其创建的所有对象被删除外，如果其他用户创建的表引用了该用户表上的主关键字或唯一关键字，或者在该表上创建了视图，DM 还将自动删除相应的引用完整性约束及视图。

5．每个数据库都有两个系统自动创建的用户 SYSDBA 和 SYSAUDITOR。他们不能被删除。

6．正在使用中的用户不能被删除。

举例说明

例 1．DBA 删除数据库 MYDB 中的用户 RU：

```
DROP USER MYDB.RU;
```

例 2．删除当前数据库中用户 RU 和他创建的所有数据库对象：

```
DROP USER RU CASCADE;
```

3.7　登录定义语句

创建新的登录。DM 数据库用户要与 DM 数据库服务器建立连接，必须使用某个已创建的登录名和口令进行登录。

语句格式

```
CREATE LOGIN   <登录名>   IDENTIFIED BY <口令>;
```

参数

1．<登录名>　指明要创建的登录名。

2．<口令>　　在 DM 初始化文件 DM.INI 中有一个 PWD_CHECK 选项，当设置为 1 时，合法的登录口令要求必须不少于 6 个字符的字符串，且是数字和字母的组合。当设置为 0 时，没有上述限制。但总长度皆不能超过 48 个字节。

语句功能

供 SYSDBA 创建新的登录。

使用说明

1．登录名在服务器中必须唯一，登录名可以与用户名相同，也可不同。登录名基本符合标准的标识符规则，但当登录名后有空格时，系统会自动删除后面的空格。

2．一个登录能为哪些用户使用，可通过登录修改语句指定。

3．系统为一个登录存储的信息主要有：登录名、缺省数据库、口令、可以使用该登录的用户及其所属数据库的一个列表。

4．缺省数据库用于用户使用该登录进行数据库连接时，没有指明登录到哪个数据库的情形，这时将会登录到缺省数据库，参见 3.6 节登录修改语句中的关于缺省数据库的说明。

5．使用该登录的用户对登录的缺省数据库及其它数据库的访问权取决于该用户对这些数库的访问权。

6．登录口令以密文形式存储。

7．DM 系统默认有一个登录口令的有效期。参见 3.6 节登录修改语句中的关于登录口令有效期的说明。

8．如果用户在登录有效期已过时登录服务器，或者登录时多次输错口令可能导致的问题参见 3.6 节登录修改语句中的相关说明。

9．DM 系统事先已经创建了两个系统登录 SYSDBA 和 SYSAUDITOR，初始口令分别为 SYSDBA、SYSAUDITOR。每个数据库都有两个系统自动创建的用户 SYSDBA 和 SYSAUDITOR。系统已指定用户 SYSDBA（或 SYSAUDITOR）可使用登录 SYSDBA（或 SYSAUDITOR）。

举例说明

例 创建登录名为 RU、口令为 RUPASS 的登录。

```
CREATE LOGIN RU IDENTIFIED BY RUPASS;
```

3.8　登录修改语句

修改登录信息。

语句格式

```
ALTER    LOGIN    <登录名>    <登录修改动作>;

<登录修改动作> ::=
        IDENTIFIED BY <登录口令>
        | DEFAULT DATABASE <缺省数据库名>
        | ADD USER <用户名 1> AT <数据库名 1>
        | DROP USER <用户名 2> AT <数据库名 2>
```

参数

1．<登录名>　　指定操作的登录的名称。

2．<登录口令>　　指重新指定的口令。

3．<缺省数据库名>　指明登录的缺省数据库的名称，参见登录定义语句的说明。

4．<用户名 1>　　往可以使用该登录的用户列表中增加新用户的用户名。

5．<数据库名 1>　　<用户名 1>指定的用户所属的数据库名。

6．<用户名 2>　　从可以使用该登录的用户列表中要进行除名的用户名。

5．<数据库名 1>　　<用户名 2>指定的用户所属的数据库名。

语句功能

供 SYSDBA 修改登录信息。其中各种<登录修改动作>的功能如下：

DENTIFIED BY <登录口令>

用于修改指定登录的口令。

DEFAULT DATABASE <缺省数据库名>

指定登录的缺省数据库。

ADD USER <用户名 1> AT <数据库名 1>

DM 系统为每个登录维护一个可以使用该登录的用户及其所属数据库的列表，

本子句的功能是将指定数据库中的指定用户（该用户必须已经存在）增加到该列表中（该过程有时称为对该登录的“用户指派”），以便以后该用户可以使用该登录与数据库服务器进行连接。

DROP USER <用户名 2> AT <数据库名 2>

DM 系统为每个登录维护一个可以使用该登录的用户及其所属数据库的列表，本子句的功能是将指定数据库中的指定用户从该列表中删除（要删除的用户在删除前应在该列表中），以后该用户就不能使用该登录与数据库服务器进行连接。

使用说明

只有 SYSDBA 用户才能使用本语句。

登录 SYSAUDITOR 的信息只能由 SYSAUDITOR 修改。

缺省数据库用于用户使用该登录进行数据库连接时，没有指明登录到哪个数据库的情形，这时将会登录到缺省数据库。创建登录时，缺省数据库默认为 SYSTEM，以后可以使用本语句另行指定登录的缺省数据库。

DM 系统事先已经创建了两个系统登录 SYSDBA 和 SYSAUDITOR，初始口令分别为 SYSDBA、SYSAUDITOR。每个数据库都有两个系统自动创建的用户 SYSDBA 和 SYSAUDITOR。系统已指定用户 SYSDBA（或 SYSAUDITOR）可使用登录 SYSDBA（或 SYSAUDITOR）。这两个登录与其同名用户的指派关系也是系统确定的，不能改动。

一个登录可以对应不同数据库中的多个用户，但在同一个库中至多只能有一个用户指派，用户名与登录名并不要求相同。相应地，该登录对数据库的访问权限取决于它在该库上用户指派的访问权限。

如果登录在某个库上没有用户指派，则无法访问该库。只有 SYSTEM 库是例外。如果登录在 SYSTEM 库上没有用户指派时，系统自动为其分配一个 GUEST 的用户指派，利用该用户指派，登录可以在 SYSTEM 库上进行简单的系统表访问，却不能作进一步的操作。

DM 系统默认一个登录的有效期（有时称登录口令有效期）为 3 个月。如有需要，可以请系统管理员修改该有效期（可用 JManager 工具修改。系统使用内部函数 SET_USER_EXPIRED_TIME 来修改登录有效期）。

如果使用有效期已过的登录试图登录服务器，服务器会报错；再次用此登录，此登录将被系统锁住，用户应该联系系统管理员为其登录解封并设置新的登录有效期。

在系统文件 DM.INI 中设定了允许用户登录时输错口令的次数，默认为 3 次。若多次输错，登录将被锁住。请向系统管理员寻求帮助。

登录信息（如登录口令和缺省数据库）的修改对所有使用该登录的用户都有效，所以修改口令或缺省数据库时应考虑及时通知其他用户的问题。对登录口令有效期的修改、对口令输入错误和口令过期的封锁及解封同样也会影响其他使用该登录的用户。

关于登录口令的合法性：在系统初始化文件 DM.INI 中，有配置项 PWD_CHECK，缺省值 PWD_CHECK = 0，表明系统不要求检查登录口令的合法性。如果要求系统检查登录口令的合法性，可以使用系统提供的系统函数 set_para_value 设置 PWD_CHECK 的值为 1。此时，合法的登录口令要求必须不少于 6 个字符的字符串，必须是数字和字母的组合。

举例说明

假定登录 RU 的口令为 PR，

例 1 把登录 RU 的口令更改为 NEWPR，更改口令的语句如下。

```
ALTER LOGIN RU IDENTIFIED BY NEWPR;
```

例 2 把该登录的缺省数据库改为数据库 MYDB：

```
ALTER LOGIN RU DEFAULT DATABASE MYDB;
```

例 3 假定数据库 MYDB 有用户 MYDB_USER1，要允许该用户使用该登录，语句如下：

```
ALTER LOGIN RU ADD USER MYDB_USER1 AT MYDB;
```

例 4 假定数据库 MYDB 用户 MYDB_USER1 已经被允许使用登录 RU，则不再允许该用户使用登录 RU 的语句如下：

```
ALTER LOGIN RU DROP USER MYDB_USER1 AT MYDB;
```

3.9　登录删除语句

删除登录。

语句格式

```
DROP LOGIN <登录名> ;
```

参数

<登录名>　指明被删除的登录的名字。

语句功能

供 SYSDBA 删除指定登录。

使用说明

仅 SYSDBA 才能删除登录。

DM 系统自动创建的两个系统登录 SYSDBA 和 SYSAUDITOR 不能被删除。

删除登录不会同时删除允许使用该登录的用户。但登录被删除后，所有用户都不能再使用该登录。

正在使用中的登录不能被删除。

举例说明

例 1. 删除登录 RU 的语句（由用户 SYSDBA 执行）：

```
DROP LOGIN RU;
```

3.10　模式定义语句

模式定义语句创建一个架构，并且可以在概念上将其看作是包含表、视图和权限定义的对象。在 DM 中，一个用户可以创建多个模式，一个模式中的对象（表、视图）可以被多个用户使用。一个模式只作用于一个数据库，不同的数据库允许有同名模式。

系统为每一个用户自动建立了一个与用户名同名的模式作为默认模式，用户还可以用模式定义语句建立其它模式。

语句格式

```
CREATE SCHEMA <模式名> [AT <数据库名>] [AUTHORIZATION <用户名>]
                  [<基表定义> |<基表修改> |<索引定义>| <视图定义>| <存储模块
                  定义>|<触发器定义>|<特权定义>];
```

参数

1．<模式名>　　　创建模式的名字。
2．<数据库名>　　模式所属的数据库名。
3．<基表定义>　　建表语句。
4．<基表修改>　　基表修改语句。
5．<索引定义>　　索引定义语句。
6．<视图定义>　　建视图语句。
7．<存储模块定义>　存储模块（存储过程或函数）定义语句。
8．<特权定义>　　授权语句。

语句功能

供具有 DBA 或 CREATE SCHEMA 权限的用户在指定数据库中定义模式。

使用说明

1．可以用 AT <数据库名>指定模式所在的数据库，若不指定，则当前数据库为模式所在的数据库。

2．<模式名>不可与其所在数据库中其它模式名相同；

3．AUTHORIZATION <用户名>标识了拥有该模式的用户；它是为其他用户创建模式时使用的。

4．使用该语句的用户必须具有 DBA 或 CREATE SCHEMA 权限；

5．定义模式时，用户可以用单条语句同时建多个表、视图，同时进行多项授权；模式一旦定义，该用户所建基表、视图等均属该模式，其它用户访问该用户所建立的基表、视图等均需在表名、视图名前冠以模式名，而建表者访问自己所建表、视图时模式名可省，这时系统自动以用户名作为模式名；

6．模式未定义之前，其它用户访问该用户所建的基表、视图等均需在表名前冠以建表者名。

举例说明

例 下面是用户 USER1 在库 MYDB 中建立模式的例子，其中 RU 是库 MYDB 中的一个用户：

```
CREATE   SCHEMA   SCHUSER1 AT "mydb" AUTHORIZATION SYSDBA;
CREATE   TABLE STAFF
(
    EMPNUM     CHAR(3) NOT NULL UNIQUE,
    EMPNAME    CHAR(20),
    GRADE DECIMAL(4),
    CITY CHAR(15)
);
CREATE VIEW STAFFV1 AS SELECT * FROM STAFF WHERE   GRADE>=12;
GRANT ALL PRIVILEGES ON STAFF TO RU;
```

用户 USER1 在库 MYDB 中建立了模式 SCHUSER1，在模式 SCHUSER1 里建了表 STAFF 和基于 STAFF 表的视图 STAFFV1，并且将表 STAFF 的所有权限给予库 MYDB 中的用户 RU。

3.11　模式删除语句

在 DM 系统中，允许用户撤销整个模式。

语句格式

```
DROP SCHEMA [<数据库名>.]<模式名> [RESTRICT | CASCADE] ;
```

参数

<模式名>　　指要删除的模式名。

<数据库名>　模式所属的数据库名。

语句功能

供具有 DBA 或该模式的所有者删除模式。

使用说明

1．<模式名>必须是指定数据库（若无 AT <数据库名>则指当前数据库）中已经存在的模式；

2．用该语句的用户必须具有 DBA 权限或是该模式的所有者；

3．若使用 CASCADE，则将整个模式和依赖它的对象都删除。否则，只有模式不被他人引用时方可删除。

举例说明

例 以 SYSDBA 身份登录数据库后，删除 MYDB 库中模式 SCHUSER1。

```
DROP SCHEMA    MYDB.SCHUSER1 CASCADE;
```

3.12　基表定义语句

用户数据库建立后，就可以定义基表来保存用户数据的结构。需指定如下信息：

1．表名、表所属的数据库名、表所属的模式名。

2．列定义。

3．完整性约束。

语句格式

```
CREATE [ GLOBAL TEMPORARY | LOCAL TEMPORARY ]
      TABLE <表名定义>
      (<列定义> {,<列定义>} {,<表级完整性约束>})
      [<STORAGE 子句>];

<表名定义> ::=
      [<模式名>.] <表名> [AT <数据库名>]
```

```
<列定义> ::=
        <列名> <数据类型>
        [IDENTITY [(<种子>,<增量>)]]
        [DEFAULT <列缺省值表达式>]
        [<列级完整性约束> {,<列级完整性约束>}] [<约束属性>]

<列级完整性约束> ::=
        [CONSTRAINT <约束名>]
        [NOT] NULL |
        PRIMARY KEY | [NOT] CLUSTER PRIMARY KEY | UNIQUE |
        <引用约束>|
        CHECK (<检验条件>)

<引用约束> ::= REFERENCES [PENDANT] [<模式名>.]<表名>(<列名>[,<列名>]…)
[WITH INDEX] [MATCH <匹配类型>] <引用触发动作>

<匹配类型> ::= FULL | PARTIAL

<引用触发动作> ::=
         [<UPDATE 规则>] [<DELETE 规则>] |
         [<DELETE 规则>] [<UPDATE 规则>]

<UPDATE 规则> ::= ON UPDATE <引用动作>

<DELETE 规则> ::= ON DELETE <引用动作>

<引用动作> ::= CASCADE | SET NULL | SET DEFAULT | NO ACTION

<约束属性> ::= [<约束检查时间>][<延迟选项>] | [<延迟选项>][<约束检查时间>]

<延迟选项> ::= [NOT] DEFERRABLE

<约束检查时间> ::= INITIALLY DEFERRABLE | INITIALLY IMMEDIATE

<表级完整性约束> ::=
        [CONSTRAINT <约束名>]
        PRIMARY KEY (<列名> {,<列名>}) |
        [NOT] CLUSTER PRIMARY KEY (<列名> {,<列名>}) |
        UNIQUE(<列名> {,<列名>})] |
                    FOREIGN KEY (<列名>{,<列名>}) <引用约束>|
```

```
        CHECK <检验条件>
        [<表级完整性约束> {,<表级完整性约束>}] [<约束属性>]

    <STORAGE 子句> ::= STORAGE(<STORAGE 项> {,<STORAGE 项>})

    <STORAGE 项> ::=
        INITIAL <初始簇数目> |
        NEXT <下次分配簇数目> |
        MINEXTENTS <最小保留簇数目> |
        ON <文件组名> |
        FILLFACTOR <填充比例>
```

参数

1．<模式名>　　指明该表属于哪个模式，缺省为当前模式。

2．<表名>　　指明被创建的基表名。

3．<数据库名>　　指明基表所在的数据库名。

4．<列名>　　指明基表中的列名。

5．<数据类型>　　指明列的数据类型。

6．<列缺省值表达式>　　如果随后的 INSERT 语句省略了插入的列值，那么此项为列值指定一个缺省值。

7．<列级完整性约束定义>中的参数：

（1）NULL　　指明指定列可以包含空值，为缺省选项。

（2）NOT NULL　　指明指定列不可以包含空值。

（3）UNIQUE　　指明指定列作为唯一关键字。

（4）PRIMARY KEY　指明指定列作为基表的主关键字。

（5）CLUSTER PRIMARY KEY　指明指定列作为基表的聚集索引主关键字。

（6）NOT CLUSTER PRIMARY KEY　指明指定列作为基表的非聚集索引主关键字。

（7）REFERENCES　指明指定列的引用约束，如果有 WITH INDEX 选项，则为外键约束建立索引，否则不建立索引，通过其他内部机制保证约束正确性。

（8）CHECK　　指明指定列必须满足的条件。

8．<表级完整性约束定义>中的参数：

（1）UNIQUE　　指明指定列或列的组合作为唯一关键字。

（2）PRIMARY KEY　指明指定列或列的组合作为基表的主关键字。指明 CLUSTER，表明是主关键字上聚集索引；指明 NOT CLUSTER，表明是主关键字上非聚集索引。

（3）FOREIGN KEY　指明表级的引用约束，如果使用 WITH INDEX 选项，则为外键约束建立索引，否则不建立索引，通过其他内部机制保证约束正确性。

（4）CHECK　　指明基表中的每一行必须满足的条件。

语句功能

供 DBA 或具有 CREATE_TABLE 权限的用户定义基表。

使用说明

1．<表名>指定了所要建立的基表名。在一个<模式>中，<基表名>、<视图名>均不相同。如果<模式名>缺省，则缺省为当前模式。若表名的首字符为'#'，则表示该表为一个临时表，只在一个会话中有效，当一个会话结束，该临时表被自动删除，同时临时表不支持多媒体数据类型。

2．普通表的表名最大长度为 128 个字符，但是含有多媒体的只能有 122 个字符，即 128-DMBLOB 串的长度

3．在建基表时，可以指定基表建在某一数据库。若<数据库名>缺省，为当前数据库。

4．所建基表至少要包含一个<列名>指定的列，在一个基表中，各<列名>不得相同。一张基表中至多可以包含 1024 列。

5．<DEFAULT 子句>指定了列的缺省值。

6. 如果未指明 NOT NULL，也未指明<DEFAULT 子句>，则隐含为 DEFAULT NULL。

7．自增列不能使用<DEFAULT 子句>。

8．<列缺省值表达式>的数据类型必须与本列的<数据类型>一致。

9．如果列定义为 NOT NULL，则其<列缺省值表达式>不能将该列指定为 NULL。

10．在建立基表时，还可定义与该表有关的完整性约束，如唯一约束、引用约束、检验约束、聚集约束等。如果完整性约束只涉及当前正在定义的列，则既可定义成列级完整性约束，也可以定义成表级完整性约束；如果完整性约束涉及到该基表的多个列，则只能在语句的后面定义成表级完整性约束。定义与该表有关的列级或表级完整性约束时，可以用 CONSTRAINT<约束名>子句对约束命名，系统中相同模式下的约束名不得重复。如果不指定约束名，系统将为此约束自动命名。经定义后的完整性约束被存入系统的数据字典中，用户操作数据库时，由 DBMS 自动检查该操作是否违背这些完整性约束条件。

11．唯一性约束定义用于指明该表中的当前列或哪几列要受唯一性约束。表定义语句可显式地指定一个主关键字。如果主关键字由多个列组成, 则称该主关键字为组合主关键字。

12．引用约束定义用于指明该表中的某些列与其它表中列的对应关系，即在引用约束定义中应指明引用列及被引用的表名和列名。被引用列必须是主关键字或受唯一性约束的列。如果在被引用的表名后未指出被引用列名，则隐含表示要以被引用表的主关键字作被引用列。建表者拥有对每个被引用列的 REFERENCES 权限。

13．DM_SQL 语言为用户提供了检验约束定义，让用户可针对某一应用环境的特定需求定义有关的完整性约束条件。<检验条件>中的每个列名必须是本表中定义的列, 但列的类型不得为多媒体数据类型，<检验条件>不应包含子查询、集函数。

14．DM_SQL 语言支持空值的概念，空值即为未知的值，没有大小，不可比较。除关键字列外，其它列可以取空值，也可规定它不可取空值，不可取空值的列要用 NOT NULL 进行说明。

15．可以使用 STORAGE 子句指定表的存储信息：

（1）初始簇数目：指建立表时分配的簇个数，必须为整数，最小值为 1，最大值为 256，缺省为 1。

（2）下次分配簇数目：指当表空间不够时，从数据文件中分配的簇个数，必须为整数，最小值为 1，最大值为 256，缺省为 1。

（3）最小保留簇数目：当删除表中的记录后，如果表使用的簇数目小于这个值，就不再释放表空间，必须为整数，最小值为 1，最大值为 256，缺省为 1。

（4）文件组名：在指定的文件组上建表，文件组必须已存在, 缺省为 PRIMARY 文件组。

（5）填充比例：指定存储索引数据时每个索引页的充满程度，取值范围从 0 到 100。默认值为 0，等价于 100，表示全满填充。创建索引时，填充比例的值越低，可由新索引项使用的空间就越多。

16．记录的列长度总和不超过块长的一半，varchar 数据类型后的长度是指数据定义长度，实际是否越界还需要判断实际记录的长度，而 char 类型后的长度是实际数据长度。与此类似的还有 VARBINARY 和 BINARY 数据类型。因此，对于 16K 的块，可以定义 CREATE TABLE TEST(C1 VARCHAR(8000),C2 INT)，但是不能定义 CREATE TABLE TEST(C1 CHAR(8000)，C2 INT)。

举例说明

例 1 首先回顾一下第二章中定义的四个基表：厂商登记、商品登记、商场登记和供货登记。它们均是将唯一性约束、引用约束和检验约束以表级完整性约束定义的格式写出的。由于这四张表的这些约束均只对单个列，所以也可以用列级完整性约束定义的格式写出。假定当前数据库为 CW，用户为 USER1，下面以商品登记表为例进行说明。

```
CREATE   TABLE   商品登记
( 商品编号     CHAR(8)   NOT NULL PRIMARY   KEY,
    商品名        CHAR(10),
    价格          NUMERIC(10,2),
    仓库地址      CHAR(10)   CHECK(仓库地址   IN   ('北京', '上海', '武汉
',NULL)),
    厂商编号     CHAR(5)   REFERENCES   厂商登记(厂商编号)
);
--注：该语句的执行需在“厂商登记”已经建立的前提下
```

系统执行建表语句后，就在数据库中建立了相应的基表，并将有关基表的定义及完整性约束条件存入数据字典中。需要说明的是，由于被引用表要在引用表之前定义，本例中的厂商登记表被商品登记表和供货登记表引用，商场登记表也被供货登记表引用，所以，这里应先定义厂商登记表和商场登记表，再定义商品登记表，最后定义供货登记表，否则就会出错。

例 2 建表时指定存储信息，表 SUTDENTS 建立在文件组 FG_STUDENT 中，初始簇大小为 5，最小保留簇数目为 5，下次分配簇数目为 2，填充比例为 85。

```
CREATE  TABLE  STUDENTS
        ( ID INT  NOT NULL PRIMARY KEY,
         NAME       CHAR(10)
        )
        STORAGE
```

```
            ( INITIAL    5,
             MINEXTENTS 5,
             NEXT        2,
             ON      FG_STUDENT,
             FILLFACTOR   85
            );
```

3.13　基表修改语句

为了满足用户在建立应用系统的过程中需要调整数据库结构的要求，DM 系统提供基表修改语句，对基表的结构进行全面的修改，包括修改基表名、列名、增加列、删除列、修改列类型、增加表级约束、删除表级约束、设置列缺省值、设置触发器状态等一系列修改。

语句格式

```
ALTER TABLE [[<数据库名>.] <模式名>.]<表名> <修改表定义语句>;

<修改表定义语句> ::= <修改表定义子句> {,<修改表定义子句>}

<修改表定义子句> ::=
        MODIFY <列定义> |
        ADD [COLUMN] <列定义> |
        ADD [COLUMN] (<列定义>{,<列定义>}) |
        DROP [COLUMN] <列名> [RESTRICT | CASCADE] |
        ADD [CONSTRAINT [<约束名>]] <表级完整性约束> |
        DROP CONSTRAINT <约束名> [RESTRICT | CASCADE] |
        ALTER [COLUMN] <列名> SET DEFAULT <列缺省值表达式>|
        ALTER [COLUMN] <列名> DROP DEFAULT |
        ALTER [COLUMN] <列名> RENAME TO <列名> |
        RENAME TO <表名> |
        ENABLE ALL TRIGGERS |
        DISABLE ALL TRIGGERS

<列定义> ::=
        <列名> <数据类型>
        [DEFAULT <列缺省值表达式>]
        [<列级完整性约束>[,<列级完整性约束>]… ] [<约束属性>]

<列级完整性约束> ::= [CONSTRAINT <约束名>]<列级约束>
```

```
<列级约束> ::=
        [NOT] NULL |
        UNIQUE |
        PRIMARY KEY |
        [NOT] CLUSTER PRIMARY KEY |
        <引用约束> |
        CHECK (<布尔表达式>)

<引用约束> ::= REFERENCES [PENDANT] [[<数据库名>.] <模式名>.]<表名> (<列名>[,<列名>]…) [WITH INDEX] [MATCH <匹配类型>] <引用触发动作>

<匹配类型> ::= FULL | PARTIAL

<引用触发动作> ::=
        [<UPDATE 规则>] [<DELETE 规则>] |
        [<DELETE 规则>] [<UPDATE 规则>]

<UPDATE 规则> ::= ON UPDATE <引用动作>

<DELETE 规则> ::= ON DELETE <引用动作>

<引用动作> ::= CASCADE | SET NULL | SET DEFAULT | NO ACTION

<约束属性> ::= [<约束检查时间>][<延迟选项>] | [<延迟选项>][<约束检查时间>]

<延迟选项> ::= [NOT] DEFERRABLE

<约束检查时间> ::= INITIALLY DEFERRABLE | INITIALLY IMMEDIATE

<表级完整性约束> ::= [CONSTRAINT [<约束名>]] <表级约束> [<约束属性>]

<表级约束> ::=
        PRIMARY KEY (<列名>[,<列名>]…) |
        [NOT] CLUSTER PRIMARY KEY |
        UNIQUE(<列名>[,<列名>]…) ] |
        FOREIGN KEY (<列名>[,<列名>]…) <引用约束> |
        CHECK <布尔表达式>
```

参数

1．<数据库名>　指明被操作的基表属于哪个数据库，缺省为当前数据库。
2．<模式名>　　指明被操作的基表属于哪个模式，缺省为当前模式。

3．<表名>　　　指明被操作的基表的名称。
4．<列名>　　　指明修改、增加或被删除列的名称。
5．<数据类型>　指明修改或新增列的数据类型。
6．<列缺省值>　指明新增/修改列的缺省值，其数据类型与新增/修改列的数据类型一致。
7．<列级完整性约束>

（1）NULL　　指明指定列可以包含空值，为缺省选项。

（2）NOT NULL　　指明指定列不可以包含空值。

（3）UNIQUE　　指明指定列作为唯一关键字。

（4）PRIMARY KEY　　指明指定列作为基表的主关键字。

（5）CLUSTER PRIMARY KEY　　指明指定列作为基表的聚集索引主关键字。

（6）NOT CLUSTER PRIMARY KEY　　指明指定列作为基表的非聚集索引主关键字。

（7）REFERENCES　　指明指定列的引用约束，如果有 WITH INDEX 选项，则为外键约束建立索引，否则不建立索引，通过其他内部机制保证约束正确性。

（8）CHECK　　指明指定列必须满足的条件。

8．<表级完整性约束定义>

（1）UNIQUE　　指明指定列或列的组合作为唯一关键字。

（2）PRIMARY KEY　　指明指定列或列的组合作为基表的主关键字。

（3）CLUSTER PRIMARY KEY　　指明指定列或列的组合作为基表的聚集索引主关键字。

（4）NOT CLUSTER PRIMARY KEY　　指明指定列或列的组合作为基表的非聚集索引主关键字。

（5）FOREIGN KEY　　指明表级的引用约束。

（6）CHECK　　指明基表中的一行必须满足的条件。

语句功能

供拥有 DBA 权限的用户或该表的建表者对表的定义进行修改，修改包括：

（1）修改一列的数据类型、精度、刻度，设置列上的 DEFAULT,NOT NULL,NULL；
（2）增加一列及该列上的列级约束；
（3）删除一列；
（4）增加表上的约束；
（5）删除表上的约束；
（6）表名/列名的重命名；
（7）修改触发器状态。

使用说明

1．使用 MODIFY COLUMN 时，要更改的列不能是：

（1）用于索引的列；
（2）用于 CHECK 或 UNIQUE 约束的列；
（3）有缺省值的列。

2．使用 MODIFY COLUMN 子句只能修改列上的 NULL，NOT NULL 约束；如果某列现有的值均非空，则允许添加 NOT NULL；只要是不属于关键字的列，都能设定为

NULL；自增列不允许被修改。

3．使用 MODIFY COLUMN 修改数据类型时，若该表中无元组，则可任意修改其数据类型、长度、精度或量度；若表中有元组，则系统会尝试修改其数据类型、长度、精度或量度，如果修改不成功，则会报错返回。

特殊说明：无论表中有、无元组，多媒体数据类型和非多媒体数据类型都不能相互转换。

4．使用 ADD COLUMN 时，新增列名之间、新增列名与该基表中的其它列名之间均不能重复。若新增列跟有缺省值，则已存在的行的新增列值是其缺省值。添加新列对于任何涉及表的视图定义或约束定义没有作用。例如：如果用“*”为一个表创建一个视图，那么后加入的新列不会自动地加入视图中，只能重新创建此视图。

5．使用 ADD COLUMN 时，还有以下限制条件：

（1）列定义中如果带有列约束，只能是对该新增列的约束；列级约束可以带有约束名，系统中同一模式下的约束名不得重复，如果不带约束名，系统自动为此约束命名。

（2）如果表上没有元组，列可以指定为 NOT NULL；如果表上有元组，该列可以指定同时有 DEFAULT 和 NOT NULL, 除此之外，不能指定为 NOT NULL。

（3）该列可指定为 CHECK。

（4）该列可指定为 FOREIGN KEY。

6．ADD CONSTRAINT 子句用于添加表级约束。表级约束包括：主键约束（PRIMARY KEY），唯一约束（UNIQUE），引用约束（REFERENCES），检验约束（CHECK）。添加表级约束时可以带有约束名，系统中同一模式下的约束名不得重复，如果不带约束名，系统自动为此约束命名。

用 ADD CONSTRAINT 子句添加约束时，对于该基表上现有的全部元组要进行约束违规验证：

（1）添加一个主键约束时，要求将成为关键字的字段上无重复值且值非空，并且表上没有定义主关键字。

（2）添加一个 UNIQUE 约束时，要求将成为唯一约束的字段上不存在重复值，但允许有空值。

（3）添加一个 REFERENCES 约束时，要求将成为外键约束的字段上的值满足该外键约束。

（4）添加一个 CHECK 约束或外键时，要求该基表中全部的元组满足该约束。

7．用 DROP COLUMN 子句删除一列有两种方式：RESTRICT 和 CASCADE。RESTRICT 方式为缺省选项，确保只有不被其他对象引用的列才被删除。无论哪种方式，表中的唯一列不能被删除。RESTRICT 方式下，下列类型的列不能被删除：被引用列、索引列、建有视图的列、有 unique 或 check 约束的列。删除列的同时将删除该列上的约束。CASCADE 方式下，将删除这一列上的引用信息和被引用信息、引用该列的视图、索引和约束；但被删除列为 CLUSTER PRIMARY KEY 类型时除外，此时不允许删除。

8．DROP CONSTRAINT 子句用于删除表级约束，表级约束包括：主键约束（PRIMARY KEY），唯一约束（UNIQUE），引用约束（REFERENCES），检验约束（CHECK）。用 DROP CONSTRAINT 子句删除一约束时，同样有 RESTRICT 和 CASCADE 两种方式。当删除主键或唯一约束时，系统自动创建的索引也将一起删除。

如果打算删除一个主键约束或一个唯一约束而它有外部约束，除非指定 CASCADE 选项，否则将不允许删除。也就是说，指定 CASCADE 时，删除的不仅仅是用户命名的约束，还有任何引用它的外部约束。

9．各个子句中都可以含有单个或多个列定义（约束定义），单个列定义（约束定义）和多个列定义（约束定义）都可以不加括号, 也可以在它们的定义外加一层括号，括号要配对。

10．具有 DBA 权限的用户或该表的建表者才能执行此操作。

举例说明

例1 商品登记表中商品照片、商品编号、厂商编号、仓库地址几列都不允许修改，分别因为：商品照片为多媒体数据类型；商品编号上定义有关键字，属于用于索引的列；厂商编号用于外键约束（包括引用列和被引用列）；仓库地址用于CHECK约束。另外，关联有缺省值的列也不能修改。而其他列都允许修改。假定当前数据库为CW，用户为USER1，如将商品名的数据类型改为varchar(8),并指定该列为NOT NULL，且缺省值为'面包'：

```
ALTER TABLE 商品登记 MODIFY 商品名 varchar(8) DEFAULT '面包' NOT NULL;
```

此语句只有在表中无元组的情况下才能成功。

例2 具有DBA权限的用户需要对供货登记表增加一列，列名为货单号，数据类型为int，并将该列定义为主关键字，值小于10000。

```
ALTER TABLE CW.USER1. 供货登记 ADD 货单号 INT PRIMARY KEY CHECK (货单号<10000);
```

如果该表上没有元组，且没有PRIMARY KEY，则可以将新增列指定为 PRIMARY KEY。表上没有元组时也可以将新增列指定为UNIQUE，但同一列上不能同时指定PRIMARY KEY和UNIQUE两种约束。

例3 具有DBA权限的用户需要对供货登记表增加一列，列名为厂商编号，数据类型为CHAR，字符长度为5，定义该列为NOT NULL，并引用厂商登记表的厂商编号。

```
ALTER TABLE CW.USER1.厂商登记 ADD 信誉等级 CHAR(5) NOT NULL
REFERENCES CW.USER1.厂商登记(厂商编号);
```

如果表上没有元组，新增列可以指定为 NOT NULL；如果表上有元组且都不为空，该列可以指定同时有 DEFAULT 和 NOT NULL，不能单独指定为 NOT NULL。

例4 具有DBA权限的用户需要删除商品登记表的商品编号一列。

```
ALTER TABLE CW.USER1.商品登记 DROP 商品编号 CASCADE;
```

删除商品编号这一列必须采用CASCADE方式，因为该列被供货登记表的商品编号引用。

例5 具有DBA权限的用户需要在表商场登记上增加UNIQUE约束，UNIQUE字段为商场名。

```
ALTER TABLE CW.USER1.商场登记 ADD CONSTRAINT CONS_3
```

```
UNIQUE(商场名);
```

用ADD CONSTRAINT子句添加约束时，对于该基表上现有的全部元组要进行约束违规验证。在这里，分为三种情况：

（1）如果表商场登记里没有元组，则上述语句一定执行成功；

（2）如果表商场登记里有元组，并且欲成为唯一约束的字段商场名上不存在重复值，则上述语句执行成功；

（3）如果表商场登记里有元组，并且欲成为唯一约束的字段商场名上存在重复值，则上述语句执行不成功，系统报错“无法建立唯一性索引”。

如果语句执行成功，用户通过查询

```
SELECT TABLEDEF('CW', 'USER1', '商场登记');
```

可以看到，修改后的商场登记的表结构显示为：

```
CREATE TABLE 商场登记 AT “CW”
(  商场编号 char(5) NOT NULL ,
   商场名   char(20),
   商场地址 char(10),
CONSTRAINT "CHECK……" CHECK(商场地址='北京' OR 商场地址='上海' OR 商场地址='武汉'),
   CONSTRAINT "CONS_3" UNIQUE(商场名),
   CONSTRAINT "INDEX……" PRIMARY KEY (商场编号)
);
```

例6 假定当前数据库为CW，库中具有DBA权限的用户需要删除商场登记表上的商场名的UNIQUE约束。当前的商场登记表结构请参见例5。

删除表约束，首先需要得到该约束对应的约束名，用户可以查询系统表SYSCONSTRAINTS，如下所示：

```
SELECT NAME FROM SYSCONSTRAINTS WHERE TABLEID =
    (SELECT ID FROM SYSTABLES WHERE NAME = '商场登记' AND SCHID=( SELECT SCHID FROM SYSSCHEMAS WHERE NAME='USER1'));
```

该系统表显示商场登记表上的所有PRIMARY KEY，UNIQUE，CHECK约束。查询得到商场名上UNIQUE约束对应的约束名，类似CHECK0134217737。

然后，可采用以下的语句删除指定约束名的约束：

```
ALTER TABLE USER1.商场登记 DROP CONSTRAINT CHECK0134217737;
```

语句执行成功。

3.14　基表删除语句

DM 系统允许用户随时从数据库中删除基表。

语句格式

```
DROP TABLE [[<数据库名>.]<模式名>.]<表名> [RESTRICT|CASCADE];
```

参数

1．<数据库名>　指明被删除基表所属的数据库，缺省为当前数据库。

2．<模式名>　　指明被删除基表所属的模式，缺省为当前模式。

3．<表名>　　　指明被删除基表的名称。

语句功能

供具有 DBA 权限的用户或该表的创建者删除基表。

使用说明

1．表删除有两种方式：RESTRICT/CASCADE 方式。其中 RESTRICT 为缺省值。如果以 CASCADE 方式删除该表，将删除表中唯一列上和主关键字上的引用完整性约束，并删除所有建立在该基表上的视图。如果以 RESTRICT 方式删除该表，要求该表上已不存在任何视图以及引用完整性约束，否则 DM 返回错误信息，而不删除该表。

2．以 RESTRICT 方式删除该表，必须要求该表上已不存在任何视图。

3．该表删除后，在该表上所建索引也同时被删除。

4．该表删除后，所有用户在该表上的权限也自动取消，以后系统中再建同名基表是与该表毫无关系的表。

举例说明

例 1 假设当前数据库为 CW，用户 USER1 需要删除供货登记表，可以用下面的语句：

```
DROP　TABLE　供货登记;
```

或

```
DROP　TABLE　CW.USER1.供货登记;
```

例 2 假设当前数据库为 CW，当前用户为用户 USER1。现要删除商场登记表。为此，必须先删除供货登记表，因为它们之间存在着引用关系，供货登记表为引用表，商场登记表为被引用表。

```
DROP TABLE 供货登记;
DROP TABLE 商场登记;
```

也可以使用 CASCADE 强制删除商场登记表，但是供货登记表仍然存在，只是删除了供货登记表两个外键约束。

```
DROP TABLE 商场登记 CASCADE;
```

3.15　全表删除语句

DM 可以从表中删除所有记录。

语句格式

```
TRUNCATE TABLE [[<数据库名>.]<模式名>.]<表名>;
```

参数

1．<数据库名>　指明表所属的数据库，缺省为当前数据库。

2．<模式名>　　指明表所属的模式，缺省为当前模式。

3．<表名>　　　指明被删除记录的表的名称。

语句功能

供具有 DBA 权限的用户或该表的创建者从表中删除所有记录。

使用说明

1．用 TRUNCATE 命令删除记录，它一次性删除指定表的所有记录，比用 DELETE 命令删除记录快。但 TRUNCATE 不会触发表上的 DELETE 触发器。

2．如果 TRUNCATE 删除的表上有被引用关系，则此语句失败。

3．TRUNCATE 不同于 DROP 命令，因为它保留了表结构及其上的约束和索引信息。

4．TRUNCATE 命令只能用来删除表的所有的记录，而 DELETE 命令可以只删除表的部分记录。

举例说明

例 假定当前数据库为 CW，具有 DBA 权限的用户删除模式 USER1 中的厂商登记表中所有的记录及其索引。

```
TRUNCATE   TABLE   USER1.厂商登记;
```

3.16　索引定义语句

为了提高系统的查询效率，DM 系统提供了索引定义语句。

语句格式

```
CREATE [OR REPLACE] [UNIQUE | BITMAP] INDEX <索引名>
    ON [[<数据库名>.]<模式名>.]<表名>(<列名>{,<列名>}) [<STORAGE 子句>];

<STORAGE 子句> ::= STORAGE(<STORAGE 项>[,<STORAGE 项>]…)

<STORAGE 项> ::=
    [INITIAL  初始簇数目] |
    [NEXT  下次分配簇数目] |
    [MINEXTENTS  最小保留簇数目] |
    [ON  文件组名])
```

参数

1．UNIQUE　　指明该索引为唯一索引。

2．BITMAP　　指明该索引为位图索引。

3．<索引名>　指明被创建索引的名称。

4．<数据库名>指明被创建索引的基表属于哪个数据库。缺省为当前数据库。

5．<模式名>　指明被创建索引的基表属于哪个模式，缺省为当前模式。

6．<表名>　　指明被创建索引的基表的名称。

7．<列名>　　指明基表中的索引列的名称，一个索引最多能有 16 列。

语句功能

供拥有 DBA 权限的用户或该表的创建者定义索引。

使用说明

1．索引列不得重复出现且数据类型不得为多媒体数据类型。

2．<索引名>不得与该模式中其它索引的名字相同。

3．可以使用 STORAGE 子句指定索引的存储信息。它的参数说明参见 CREATE TABLE 语句。

4．索引各字段值相加得到的总数据值长度不得超过 1020。

5．在下列情况下，DM 利用索引可以提高性能：

（1）用指定的索引列值来搜索记录。

（2）用索引列的顺序来存取基表。

然而，索引也会降低那些影响索引列值的命令，如 INSERT，UPDATE，DELETE 的性能，因为 DM 不但要维护基表数据还要维护索引数据。

举例说明

例 假设当前数据库为 CW，具有 DBA 权限的用户在供货登记表中，以商品编号为索引列建立索引 S1，以商场编号、商品编号、进货日期为索引列建立唯一索引 S2。

```
CREATE INDEX S1 ON CW.USER1.供货登记(商品编号);
CREATE UNIQUE INDEX S2 ON CW.USER1.供货登记(商场编号,商品编号,进货日期);
```

3.17　索引删除语句

DM 系统允许用户在建立索引后还可随时删除索引。

语句格式

```
DROP INDEX [[<数据库名>.]<模式名>.]<索引名>;
```

参数

1．<数据库名>　　指明被删除索引所属的数据库，缺省为当前数据库。

2．<模式名>　　指明被删除索引所属的模式，缺省为当前模式。

3．<索引名>　　指明被删除索引的名称。

语句功能

供拥有 DBA 权限的用户或该表的创建者删除索引。

使用说明

使用者应拥有 DBA 权限或是该表的建表者。

举例说明

例 具有 DBA 权限的用户需要删除 S1 索引可用以下语句实现：

```
DROP INDEX S1;
```

3.18　序列定义语句

序列是一个数据库实体，通过它多个用户可以产生唯一整数值，可以用序列来自动地生成主关键字值。

语句格式

```
CREATE SEQUENCE [ <模式名>.] <序列名> [AT <数据库名>]
    [INCREMENT BY <增量值>]
    [START WITH <初值>]
    [MAXVALUE <最大值>]
    [MINVALUE <最小值>]
    [CYCLE];
```

参数

1．<数据库名>　指明被创建的序列属于哪个数据库，缺省为当前数据库。

2．<模式名>　　指明被创建的序列属于哪个模式，缺省为当前模式。

3．<序列名>　　指明被创建的序列的名称。

4．<增量值>　　指定序列数之间的间隔，这个值可以是任意的 DM 正整数或负整数，但不能为 0。如果此值为负，序列是下降的，如果此值为正，序列是上升的。如果忽略 INCREMENT BY 子句，则间隔缺省为 1。

5．<初值>　　　指定被生成的第一个序列数，可以用这个选项来从比最小值大的一个值开始升序序列或比最大值小的一个值开始降序序列。对于升序序列，缺省值为序列的最小值，对于降序序列，缺省值为序列的最大值。

6．<最大值>　　指定序列能生成的最大值，如果忽略 MAXVALUE 子句，则降序序列的最大值缺省为 -1，升序序列的最大值为 9223372036854775806（0x7FFFFFFFFFFFFFFE）。序列在到达最大值之后，将返回初始值。

7．<最小值>　　指定序列能生成的最小值，如果忽略 MINVALUE 子句，则升序序列的最小值缺省为 1，降序序列的最小值为 -9223372036854775808（0x8000000000000000）。序列在到达最小值之后，将返回初始值。

语句功能

创建一个序列生成器。只有 DBA 或具有 CREATE SEQUENCE 权限的用户才能创建序列。

使用说明

1．一旦序列生成，就可以在 SQL 语句中用以下伪列来存取序列的值。

（1）CURRVAL 返回当前的序列值。

（2）NEXTVAL 如果为升序序列，序列值增加并返回增加前的值；如果为降序序列，序列值减少并返回增加前的值。

2．缺省序列：如果在序列中什么也没有指出则缺省生成序列，一个从 1 开始增量为 1 且无限上升(最大值为 2^{31}-1)的升序序列；仅指出 INCREMENT BY -1，将创建一个从-1 开始且无限下降(最小值为-2^{31})的降序序列。

举例说明

例　创建序列 ESEQ，将序列的前两个值插入表 T1 中。

```
（1）创建表 T1
     CREATE   TABLE   T1(COL1   INT);
（2）创建序列 ESEQ
     CREATE   SEQUENCE   ESEQ   INCREMENT   BY   10;
```

（3）将序列的第一个值插入表 T1 中
```
INSERT    INTO    T1    VALUES(ESEQ.NEXTVAL);
SELECT    *    FROM    T1;
```
查询结果为：表 T1 有一行，列 COL1 的值为 1

（4）将序列的第二个值插入表 T1 中
```
INSERT    INTO    T1    VALUES(ESEQ.NEXTVAL);
SELECT    *    FROM    T1;
```
查询结果为：表 T1 有两行，列 COL1 的值分别为 1，11

3.19　序列删除语句

DM 系统允许用户在建立序列后还可随时删除序列。

语句格式

```
DROP SEQUENCE [[<数据库名>.]<模式名>.]<序列名>;
```

参数

1．<数据库名>　指明被删除序列所属的数据库，缺省为当前数据库。
2．<模式名>　指明被删除序列所属的模式，缺省为当前模式。
3．<序列名>　指明被删除序列的名称。

语句功能

从数据库中删除序列生成器。

使用说明

一种重新启动序列生成器的方法就是删除它然后再重新创建，例如有一序列生成器当前值为 150，而且用户想要从值 27 开始重新启动此序列生成器，他可以：

（1）删除此序列生成器。
（2）重新以相同的名字创建序列生成器，START WITH 选项值为 27。

举例说明

例　USER1 用户需要删除序列 ESEQ，可以用下面的语句：

```
DROP    SEQUENCE    ESEQ;
```

3.20　全文索引定义语句

用户可以在指定的表的文本列上建立全文索引。

语句格式

```
CREATE CONTEXT INDEX    <索引名>    ON [[<数据库名>.]<模式名>.] <表名> (<列名>);
```

参数

1．<索引名>　指明要创建的全文索引的名称。
2．<数据库名>　指明要创建全文索引的基表属于哪个数据库，缺省为当前数据库。
3．<模式名>　指明要创建全文索引的基表属于哪个模式，缺省为当前模式。

4．<表名>　　　指明要创建全文索引的基表的名称。

5．<列名>　　　指明基表中要创建全文索引的列的名称。

语句功能

在指定的表的文本列上建立全文索引。

使用说明

1. 本语句只标记在数据库指定表的指定列上建立全文索引，并没有真正建立并保存全文索引信息。

2．全文索引必须在基表、系统表上定义，而不能在视图或临时表上定义。

3．DM 定义全文索引时，不需要像 SQL Server 那样要在表上先建立主关键字索引。

4．一条全文索引作用于表的一个文本列，每一个文本列只能创建一个全文索引，但对一个表可以在不同的列上建立多个全文索引。

5．<列名>为一文本列，类型可为 CHAR,CHARACTER,VARCHAR, LONGVARCHAR,TEXT 或 CLOB。

6．TEXT、CLOB 类型的列可存储二进制字符流数据。如果用于存储 DM 全文检索模块能识别的格式简单的文本文件（如.txt，html 等），则可为其建立全文索引。如果存储的是格式复杂的文件（如.doc 和.pdf 等），将无法识别并建立索引。

7．全文索引支持中、英、俄文等多种语言。

8．创建索引后必须用全文索引修改语句填充索引信息，才能进行全文检索。

3.21　全文索引修改语句

重建式填充/更新全文索引。全文索引不能自动随数据的操纵永远保持最新状态，必须定期更新；若表数据变化而没有更新索引，可能引起查询结果不正确。

语句格式

```
ALTER CONTEXT INDEX <索引名> ON [[<数据库名>.]<模式名>.] <表名> REBUILD;
```

参数

1．<索引名>　　指明被操作的全文索引的名称。

2．<数据库名>　指明被操作的全文索引属于哪个数据库，缺省为当前数据库。

3．<模式名>　　指明被操作的全文索引属于哪个模式，缺省为当前模式。

4．<表名>　　　指明被操作的基表的名称。

语句功能

将表中指定建立全文索引的列逐元组取出，根据已有词库逐字逐词分析，建立全文索引，获取该字/词出现的位置信息（元组＋元组中的位置），保存在相应的数据库文件中。

使用说明

1．CREATE CONTEXT INDEX 后，需要调用此语句，之后才能进行全文检索。这时该修改语句起到填充全文索引信息的作用。

2．当该列数据大量更新后，为了对更新后的数据进行检索，需要再次调用该语句

重建全文索引信息，以确保查询的正确性。

3．DM 服务器启动时，将自动导入中英文字库。

4．目前不提供增量式更新全文索引。

3.22　全文索引删除语句

删除全文索引。

语句格式

```
DROP CONTEXT INDEX　<索引名> ON　[[<数据库名>.]<模式名>.] <表名>;
```

参数

1．<索引名>　　指明被操作的全文索引的名称。

2．<数据库名>　指明被操作的全文索引属于哪个数据库，缺省为当前数据库。

3．<模式名>　　指明被操作的全文索引属于哪个模式，缺省为当前模式。

4．<表名>　　　指明被操作的基表的名称。

语句功能

删除全文索引，包括删除数据字典中的相应信息和全文索引内容。

使用说明

1．除了该语句可删除全文索引外，当数据库模式发生如下改变时，系统将自动调用全文索引删除模块：

2．删除数据库时，删除库中的所有全文索引；

3．删除表时，删除表上的所有全文索引；

4．当建立了全文索引的列的类型改变时，总是删除列上的全文索引；

5．当建立了全文索引的列被删除时，总是删除列上的全文索引。

第 4 章　数据查询语句和全文检索语句

数据查询是数据库的核心操作，DM_SQL 语言提供了功能很强的查询语句供用户使用。由于对数据库的其它操作如插入、删除、修改等也均要以查询定位作为前提，因此掌握查询语句的使用是很重要的。

在 DM_SQL 语言中，有的语句格式中就包含有查询语句，如视图定义语句、游标定义语句等。为了区别，我们将出现在其它语句中的查询语句称查询说明。

由于查询语言使用方式灵活多变，使用得当会大大提高系统的查询效率。为方便用户的使用，本节将对常用的形式分别举例说明，各例中所用基表及各基表中预先装入的数据参见第 2 章。各例的建表者均为具有 RESOURCE 权限的用户 USER1，并假定当前数据库为 CW。

DM 提供全文检索服务，利用全文索引，可快速搜索包含某个词或某一组词的记录。全文检索是一种特殊的查询，在本章 4.8 节另作介绍。

下面首先介绍一下查询语句的语法格式：

语法格式

```
SELECT [TOP N1 [PERCENT | , N2 ]] <选择列表>
    FROM <表引用> [{,<表引用>}]
    [<WHERE 子句>]
    [<GROUP BY 子句>]
    [<HAVING 子句>]
    {,UNION [ALL]<查询表达式>}
    [<ORDER BY 子句>]
    [FOR UPDATE][OF<列名>{,<列名>}];

<选择列表> ::= [ALL | DISTINCT]
        [[[<数据库名>.]<模式名>.]<基表名> | <视图名> .] * | <值表达式> [AS <列别名>]
        {,[[[<数据库名>.]<模式名>.]<基表名> | <视图名>.] * | <值表达式> [AS <列别名>]}

<表引用> ::= [[<数据库名>.]<模式名>.]<基表名> | <视图名> <相关名> |
    <表子查询> [ AS <相关名>] [<派生列表>]|
    <连接表>

<表子查询> ::= ( <查询表达式>)

<派生列表> ::= <列名> [ { , <列名> }]
```

```
<连接表>::=<交叉连接>|<限定连接>|<左括号><连接表><右括号>

<交叉连接>::=<表引用> CROSS JOIN <表引用>

<限定连接>::=<表引用>[NATURAL][<连接类型>] JOIN <表引用>[<连接条件>]

<连接类型>::=INNER|<外连接类型>[OUTER]

<外连接类型>::=LEFT|RIGHT|FULL

<连接条件>::=ON <搜索条件>

<WHERE 子句> ::= WHERE <搜索条件>

<GROUP BY 子句> ::= <列名> | <值表达式>{,<列名> | <值表达式>}

<HAVING 子句> ::= HAVING <搜索条件>

<ORDER BY 子句> ::= ORDER BY
        <无符号整数> | <列说明> | <值表达式> [ASC | DESC]
        {,<无符号整数> | <列说明> | <值表达式> [ASC | DESC]}
```

参数

1．ALL　　　　返回所有被选择的行，包括所有重复的拷贝，缺省值为 ALL。

2．DISTINCT　从被选择出的具有重复行的每一组中仅返回一个这些行的拷贝。对于集合算符：UNION，缺省值为 DISTINCT。

3．<数据库名>　被选择的表和视图所属的数据库，缺省为当前数据库。

4．<模式名>　　被选择的表和视图所属的模式，缺省为当前模式。

5．<基表名>　　被选择数据的基表的名称。

6．<视图名>　　被选择数据的视图的名称。

7．*　　　　　指定对象的所有的列。

8．<值表达式>　从在 FROM 子句中列出的表、视图中选择一表达式，通常是基于列的值。如果表、视图具有指定的用户名限定，则列要用用户名限定。

9．<列别名>　　为列表达式提供不同的名称，使之成为列的标题，列别名不会影响实际的名称，别名在该查询中被引用。

10．<相关名>　给表、视图提供不同的名字，经常用于求子查询和相关查询的目的。

11．<列名>　　指明列的名称。

12．<WHERE 子句>　限制被查询的行必须满足条件，如果忽略该子句，DM 从在 FROM 子句中的表、视图中选取所有的行。

13．<HAVING 子句>　限制所选择的行组所必须满足的条件，缺省为恒真，即对所

有的组都满足该条件。

14．<无符号整数>　　指明了要排序的<值表达式>在 SELECT 后的序列号。

15．<列说明>　　排序列的名称。

16．ASC　　指明为升序排列，缺省为升序。

17．DESC　　指明为降序排列。

18．FOR UPDATE　　指明锁住选择出的行。

使用说明

1．<选择列表>中最多可包含 1024 个查询项，且查询记录的长度限制不能超过块长的一半。

2．<FROM 子句>中最多可引用 50 张表。

4.1　单表查询

本节着重于较简单的查询语句——单表查询，即 SELECT 语句仅从一个表/视图中检索数据。以下是一个单表查询语句的语法：

```
SELECT <选择列表>
    FROM [[<数据库名>.]<模式名>.]<基表名> | <视图名> <相关名>
    [<WHERE  子句>]
    [<GROUP BY  子句>]
    [<HAVING  子句>];
```

可选项 WHERE 子句用于设置对于行的检索条件。不在规定范围内的任何行都从结果表中去除。GROUP BY 子句逻辑地将由 WHERE 子句返回的临时结果重新编组。结果是行的集合，一组内一个成组列的所有值都是相同的。HAVING 子句用于为组设置检索条件。本节将详细介绍 WHERE 子句，后两种子句将在本章第 4.5 节中详细介绍。

4.1.1　简单查询

例 查询所有商品的名字、价格及仓库地址，并消去重复。

```
SELECT　DISTINCT　商品名, 价格, 仓库地址　FROM　商品登记;
```

其中，DISTINCT 保证重复的行将从结果中去除。若允许有重复的元组，改用 ALL 来替换 DISTINCT，或直接去掉 DISTINCT 即可。

查询结果如下表 4.1.1 所示。（注：除带 Order By 的查询外，本书所示查询结果中各元组的顺序与实际输出结果中的元组顺序不一定一致。）

表 4.1.1 查询结果

商品名	价格	仓库地址
电视机	3000	武汉
电冰箱	2000	武汉
鞋_1	50	武汉

电视机	2500	上海
洗衣机	1000	上海
鞋_2	500	武汉
电视机	2800	北京
电冰箱	1900	上海

有时，用户需要查出各个列的数据，且各列的显示顺序也与基表中列的顺序相同时，为了方便用户提高工作效率，SQL 语言允许用户将 SELECT 后的<值表达式>省略为*。

```
SELECT　*　FROM　商场登记;
```

等价于：

```
SELECT　商场编号, 商场名, 商场地址　FROM　商场登记;
```

其查询结果是基表商场登记的一份拷贝，结果从略。

4.1.2　带条件查询

带条件查询是指在指定表中查询出满足条件的元组。该功能是在查询语句中使用 WHERE 子句实现的。WHERE 子句常用的查询条件由谓词和逻辑运算符组成。谓词指明了一个条件，该条件求解后，结果为一个布尔值：真、假或未知。

逻辑算符有：AND，OR，NOT。

谓词包括比较谓词(＝、＞、＜、＞＝、＜＝、＜＞)，BETWEEN 谓词、IN 谓词、LIKE 谓词、NULL 谓词、QUANTIFIED 谓词、EXISTS 谓词。

1．使用比较谓词的查询

当使用比较谓词时，数值数据根据它们代数值的大小进行比较，字符串的比较则按序对同一顺序位置的字符逐一进行比较。若两字符串长度不同，短的一方应在其后增加空格，使两串长度相同后再作比较。

例 给出价格在 500～2000 元之间的所有商品的情况。

```
SELECT　*　FROM　商品登记
WHERE　价格>=500　AND　价格<=2000;
```

查询结果如下表 4.1.2 所示。

表 4.1.2 查询结果

商品编号	商品名	价格	仓库地址	厂商编号
C0002	电冰箱	2000.00	武汉	B0A02
C0005	洗衣机	1000.00	上海	B0A05
C0006	鞋_2	500.00	武汉	B0003
C0008	电冰箱	1900.00	上海	B0A02

2．使用 BETWEEN 谓词的查询

例 给出价格在 500～2000 元之间的所有商品的情况。

```
SELECT　*　FROM　商品登记
WHERE　价格　BETWEEN　500　AND　2000;
```

此例查询与上例完全等价，查询结果如上表所示。在 BETWEEN 谓词前面可以使用

NOT，以表示否定。

3．使用 IN 谓词的查询

谓词 IN 可用来查询某列值属于指定集合的元组。

例 查询北京、上海两地的厂商名，资产总值。

```
SELECT　厂商名, 资产总值　FROM　厂商登记
WHERE　厂商地址　IN　('北京', '上海');
```

查询结果如下表 4.1.3 所示。

表 4.1.3 查询结果

厂商名	资产总值
华乐电视机厂	50
万里鞋厂	30
小鸭洗衣机厂	1000

在 IN 谓词前面也可用 NOT 表示否定。

4．使用 LIKE 谓词的查询

LIKE 谓词一般用来进行字符串的匹配。我们先用实例来说明 LIKE 谓词的使用方法。

例 1 查询生产家电产品的厂商情况(即厂商编号倒数第三个字符为“A”的厂商)。

```
SELECT　厂商编号, 厂商名, 厂商地址　FROM　厂商登记
WHERE　厂商编号　LIKE　'%A__';
```

查询结果如下表 4.1.4 所示。

表 4.1.4 例 1 的查询结果

厂商编号	厂商名	厂商地址
B0A01	华乐电视机厂	北京
B0A02	日东冰箱厂	武汉
B0A04	海天电视机厂	武汉
B0A05	小鸭洗衣机厂	上海

由上例可看出，LIKE 谓词的一般使用格式为：

```
<列名>　LIKE　<匹配字符串常数>
```

其中，<列名>必须是 CHAR 和 VARCHAR 类型的列。对于一个给定的目标行，如果指定列值与由<匹配字符串常数>给出的内容一致，则谓词结果为真。<匹配字符串常数>中的字符可以是一个完整的字符串，也可以是百分号“%”和下划线“_”，“%”和“_”称通配符。“%”代表任意字符串(也可以是空串)；“_”代表任何一个字符。

因此，上例中的 SELECT 语句将从厂商登记表中检索出厂商编号中倒数第三个字符为“A”的厂商，即生产家电产品的厂商情况。从该例我们可以看出 LIKE 谓词是非常有用的。使用 LIKE 谓词可以找到所需要的但又记不清楚的那样一些信息。这种查询称模糊查询或匹配查询。为了加深对 LIKE 谓词的理解，下面我们再举几例：

```
商品名　LIKE　'%电%'
```

如果商品名的值含有字符“电”，则该谓词取真值。

```
厂商编号　LIKE　'B_A__'
```

如果厂商编号的值由五个字符组成且第一个字符为 B，第三个字符为 A，则该谓词

取真值。

```
分类号　LIKE　'%P___'
```

如果分类号的值由四个或四个以上字符组成，且倒数第四个字符是 P，则该谓词取真值。

```
商场编号　NOT　LIKE　'A%'
```

如果商场编号的值的第一个字符不是 A，则该谓词取真值。

阅读以上的例子，读者可能就在想这样一个问题：如果<匹配字符串常数>中所含“%”和“_”不是作通配符，而只是作一般字符使用应如何表达呢？为解决这一问题，SQL 语句对 LIKE 谓词专门提供了对通配符“%”和“_”的转义说明，这时 LIKE 谓语使用格式为：

```
<列名>　LIKE　'<匹配字符串常数>'　[ESCAPE　<转义字符>]
```

其中，<转义字符>指定了一个字符，当该字符出现在<匹配字符串常数>中时，用以指明紧跟其后的“%”或“_”不是通配符而仅作一般字符使用。

例 2 查询鞋的商品编号，价格和仓库地址。

在商品登记表中，鞋有两种，一种鞋名为“鞋_1”，另一种为“鞋_2”，在此例中用 LIKE 谓词应为：

```
SELECT　商品编号, 价格, 仓库地址　FROM　商品登记
WHERE　商品名　LIKE　'鞋*__'　ESCAPE　'*';
```

在此例中，*被定义为转义字符，因而在<匹配字符串常数>中*号后的第一个下划线不再作通配符，而第二个下划线仍作通配符。

查询结果如下表 4.1.5 所示。

表 4.1.5 例 2 的查询结果

商品编号	价格	仓库地址
C0003	50.00	武汉
C0006	500.00	武汉

为避免错误，转义字符一般不要选通配符“%”、“_”或在<匹配字符串常数>中已出现的字符。

5．使用 NULL 谓词的查询

空值是未知的值。当列的类型为数值类型时，NULL 并不表示为 0；当列的类型为字符串类型时，NULL 也并不表示空串。因为 0 和空串也是确定值。NULL 只能是一种标识，表示它在当前行中的相应列值还未确定或未知，对它的查询也就不能使用比较谓词而须使用 NULL 谓词。

例 查询哪些商品暂时还不能供应(价格为空)。

```
SELECT　商品编号, 商品名　FROM　商品登记
WHERE　价格　IS NULL;
```

在 NULL 谓词前，可加 NOT 表示否定。

6. 组合逻辑

可以用逻辑算符（AND，OR，NOT）与各种谓词相组合生成较复杂的条件查询。

例 查询所有价格低于 1000 且不等于 800 的商品或商品名为电视机的商品编号及价格。

```
SELECT　商品编号,价格　FROM　商品登记
WHERE　价格<1000 AND 价格<>800 OR 商品名='电视机';
```

查询结果如下表 4.1.6 所示。

表 4.1.6 查询结果

商品编号	价格
C0003	50.00
C0006	500.00
C0001	3000.00
C0004	2500.00
C0007	2800.00

4.1.3　集函数

为了进一步方便用户的使用，增强查询能力，SQL 语言提供了多种内部集函数。集函数又称库函数，当根据某一限制条件从表中导出一组行集时，使用集函数可对该行集作统计操作。集函数按文本可分为三类：

（1）COUNT(*)

（2）相异集函数　AVG|MAX|MIN|SUM|COUNT (DISTINCT<列名>)

（3）完全集函数　AVG|MAX|MIN| COUNT|SUM([ALL]<值表达式>）

相异集函数与完全集函数的区别是：相异集函数是对表中的列值消去重复后再作集函数运算，而完全集函数是对包含列名的值表达式作集函数运算且不消去重复。

在使用集函数时要注意以下几点：

（1）集函数中的自变量不允许是集函数，即不能嵌套使用。

（2）要注意 DISTINCT 的用法。

（3）AVG、SUM 的参数必须为数值类型；MAX、MIN 的结果数据类型与参数类型保持一致；对于 SUM 函数，如果参数类型为 BYTE、BIT、SMALLINT 或 INTEGER，那么结果类型为 INTEGER，如果参数类型为 NUMERIC、DECIMAL、FLOAT、DOUBLE PRECISION 和 MONEY，那么结果类型为 DOUBLE PRECISION；COUNT 结果类型统一为 INTEGER；对于 AVG 函数，其参数类型与结果类型对应关系如下：

参数类型	结果类型
tinyint	dec(3,1)
smallint	dec(5,1)
int	dec(10,1)
bigint	bigint
float	double
double	double
dec(x,y)	dec(x1,y1)

如果 y<6，则 y1=6，否则 y1=y;如果 x<19,则 x1=x+19，否则 x1=38

下面按集函数的功能分别进行介绍。

1．求最大值集函数 MAX 和求最小值集函数 MIN

例 查询几种电冰箱的最低价格。

```
SELECT　MIN(价格)　FROM　商品登记
WHERE　商品名='电冰箱';
```

查询结果为：1900

需要说明的是：SELECT 后使用集函数 MAX 和 MIN 得到的是一个最大值和最小值，因而 SELECT 后不能再有列名出现，如果有只能出现在集函数中。如：

```
SELECT　商品名,MIN(价格)　FROM　商品登记;
```

DM 系统会报错，因为商品名是一个行集合，而最低价格是唯一值。

至于 MAX 的使用格式与 MIN 是完全一样的，希望读者自己举一反三。

2．求平均值集函数 AVG 和总和集函数 SUM

例 求武汉厂商的平均资产总值。

```
SELECT　AVG(资产总值)　FROM　厂商登记
WHERE　厂商地址='武汉';
```

查询结果为：53.33

3．求总个数集函数 COUNT

例 查询已登记厂商的家数。

```
SELECT　COUNT(*)　FROM　厂商登记;
```

查询结果为：6

由此例可看出，COUNT(*)的结果是厂商登记表中的总行数，由于主关键字不允许有相同值，因此，它不需要使用保留字 DISTINCT。

例 查询目前正供货的厂商数。

```
SELECT　COUNT(DISTINCT　厂商编号)　FROM　商品登记;
```

查询结果为：5

由于一个厂商可供应多种商品，因而此例中一定要用 DISTINCT 才能得到正确结果。

4.1.4　情况表达式

<值表达式>可以为一个<列引用>、<集函数>、<标量子查询>或<情况表达式>等等。<情况表达式>包括<情况缩写词>和<情况说明>两大类。<情况缩写词>包括函数 NULLIF 和 COALESCE，在 DM 中被划分为空值判断函数。具体函数说明请见 8.4 节。下面详细介绍<情况说明>表达式。

<CASE 情况说明>的语法和语义如下：

语法格式

```
<情况说明> ::= <简单情况> | <搜索情况>

<简单情况> ::= CASE
                    <值表达式>
                    {<简单 WHEN 子句> }
```

```
                    [<ELSE 子句>]
                    END

<搜索情况> ::= CASE
                      {<搜索 WHEN 子句>}……
                      [<ELSE 子句>]
                    END

<简单 WHEN 子句> ::= WHEN <值表达式> THEN <结果>

<搜索 WHEN 子句> ::= WHEN <搜索条件> THEN <结果>

<结果> ::= <值表达式> | NULL
```

功能

指明一个条件值。将搜索条件作为输入并返回一个标量值。

语法规则

1．在<情况说明>中至少有一个<结果>应该指明<值表达式>。

2．如果未指明<ELSE 子句>，则隐含 ELSE NULL。

3．<简单情况>中，CASE 运算数的数据类型必须与<简单 WHEN 子句>中的<值表达式>的数据类型是可比较的，且与 ELSE 子句的结果也是可比较的。

4．<情况说明>的数据类型由<结果>中的所有<值表达式>的数据类型确定。

（1）如果<结果>指明 NULL，则它的值是空值。

（2）如果<结果>指明<值表达式>，则它的值是该<值表达式>的值。

5．CASE 子句的语法中不支持 WHEN(1)用法，请使用以下语法：when(1 > 0)

6．如果在<情况说明>中某个<搜索 WHEN 子句>的<搜索条件>为真，则<情况说明>的值是其<搜索条件>为真的第一个<搜索 WHEN 子句>的<结果>的值，并按照<情况说明>的数据类型来转换。

7．如果在<情况说明>中没有一个<搜索条件>为真，则<情况表达式>的值是其显式或隐式的<ELSE 子句>的<结果>的值，并按照<情况说明>的数据类型来转换。

举例说明

例 1 建立一个基表，当表中第一列的值大于 3 返回'big'，等于 3 返回'equal'，否则返回'small'。

```
create table casetab(
 c int,
 d smallint,
 e char(10)
);
insert into casetab values(1, 100, 'one');
insert into casetab values(2, 200, 'two');
insert into casetab values(3, 300, 'three');
```

```
insert into casetab values(4, 400, 'four');
insert into casetab values(5, 500, 'five');
select   c,
         CASE
             WHEN c>3 THEN 'big'
             WHEN c=3 THEN 'equal'
             ELSE 'small'
         END
from casetab;
```

查询结果如表 4.1.7 所示：

表 4.1.7 例 1 的查询结果

C	CASE
1	small
2	small
3	equal
4	big
5	big

例 2 在上述表中寻找第一列值为 1，且第三列值为'one'的元组，找到则返回'get it'，否则返回'not found'。

```
select c,
      case
            when c = 1 and e = 'one' then 'get it'
            else 'not found'
      end
from casetab;
```

查询结果如表 4.1.8 所示：

表 4.1.8 例 2 的查询结果

C	CASE
1	get it
2	not found
3	not found
4	not found
5	not found

例 3 在上述表中选择第一列为 3 且第三列为'three'的元组返回。

```
select * from casetab
 where e in (select case
                    when c=3 then 'three'
```

```
            else 'not equal'
            end
          from casetab);
```

查询结果如表 4.1.9 所示：

表 4.1.9 例 3 的查询结果

C	D	E
3	300	three

例 4 在上述表中，若第一列值不大于 3 则修改该值为 1。

```
update casetab set c = CASE
        WHEN c>3 THEN   c
        ELSE 1
      END;
select * from casetab;
```

查询结果如表 4.1.10 所示：

表 4.1.10 例 4 的查询结果

C	D	E
1	100	one
1	200	two
1	300	three
4	400	four
5	500	five

4.2　连接查询

一个数据库的多个表之间一般都有某种内在的联系，因此，用户经常需要从不同的表中同时查询相关的信息。若一个查询同时涉及两个以上的表，则称之为连接查询。SQL 对连接的定义为：由笛卡尔积、内部连接、外部连接和联合连接得出的表。

<连接表>的语法如下：

```
<连接表>::=<交叉连接>|<限定连接>|（<连接表>）

<交叉连接>::=<表引用> CROSS JOIN <表引用>

<限定连接>::=<表引用>[NATURAL][<连接类型>] JOIN <表引用>[<连接条件>]

<连接类型>::=INNER|<外连接类型>[OUTER]

<外连接类型>::=LEFT|RIGHT|FULL
```

```
<连接条件>::=ON <搜索条件> | USING ( <连接列列名> [{,<连接列列名>}…] )
```

下面分别举例说明。

1．笛卡尔积过滤连接

例 1 建表者 USER1 查询由北京厂商提供的所有商品名和价格。

```
SELECT DISTINCT  商品名,价格    FROM    厂商登记,商品登记
WHERE    商品登记.厂商编号=厂商登记.厂商编号    AND    厂商地址='北京';
```

查询结果如下表 4.2.1 所示。

表 4.2.1 例 1 的查询结果

商品名	价格
电视机	3000.00
鞋-1	50.00
鞋-2	500.00

本例中的查询数据必须来自厂商登记和商品登记两个表。因此，应在 FROM 子句中给出这两个表的表名，在 WHERE 子句中给出连接条件(即要求两个表中厂商编号的列值相等)。当参加连接的表中出现相同列名时，为了避免混淆，可在这些列名前加表名前缀。

该例的查询结果是厂商登记和商品登记表在厂商编号列上做等值连接产生的。条件“商品登记.厂商编号=厂商登记.厂商编号”称为连接条件或连接谓词。当连接运算符为“=”号时，称为**等值连接**，使用其它运算符则称**非等值连接**。

关于此例有必要再做几点说明：

(1) 连接谓词中的列类型必须是可以比较的，但不一定要相同。例如，其中一个是整形而另一个是实型，它们均属数值类型，但不允许一个是字符串类型而另一个是数值类型。

(2) 不要求连接谓词中的列同名，虽然它们一般是同名的。

(3) 不要求连接谓词中的比较操作符只能是等号，尽管等值连接是最常用的。

(4) 不要求做连接的列出现在结果表中。

(5) WHERE 子句中可同时包含连接条件和其它非连接条件。

(6) 等值连接的结果表中可含有相同的列。如果删去一个表中的相同列，就得到自然连接的结果。自然连接可以说是最有用的连接，通常不加限定词的术语“连接”称为自然连接。

(7) 从概念上讲，可以进行两个表、三个表、……，任意多个表的连接。

最后需要说明的是：本查询语句的使用者是建表者 USER1，如果用户 USER2 具有厂商登记表和商品登记表的 SELECT 权限，也做上面同样的查询，所用查询语句的格式应为：

```
SELECT    DISTINCT    商品名,价格
FROM    USER1.厂商登记,USER1.商品登记
WHERE    USER1.商品登记.厂商编号=USER1.厂商登记.厂商编号
AND    USER1.厂商登记.厂商地址='北京';
```

也可以使用别名来简化：

```
SELECT DISTINCT  商品名,价格
FROM    USER1.厂商登记  A,USER1.商品登记  B
```

```
WHERE   B.厂商编号=A.厂商编号   AND   A.厂商地址='北京';
```

为叙述方便，本节在不作特别说明时，查询语句的使用者均为建表者 USER1。

例 2 查询如下商品的名称与价格：该商品的厂商与进货商场在同一城市。

```
SELECT   商品登记.商品编号,商品名,价格
FROM     厂商登记,商品登记,商场登记,供货登记
WHERE    厂商登记.厂商编号=商品登记.厂商编号
    AND  商品登记.商品编号=供货登记.商品编号
    AND  供货登记.商场编号=商场登记.商场编号
    AND  厂商地址=商场地址;
```

查询结果如下表 4.2.2 所示。

表 4.2.2 例 2 的查询结果

商品编号	商品名	价格
C0002	电冰箱	2000.00
C0004	电视机	2500.00

2．自然连接(NATURAL JOIN)

当遇到两个表中相同的<列名>时，连接它们。如下例所示：

假设现有如下两个表 BOOKPRICE，BOOKAUTHOR。

BOOKPRICE

BOOKNAME	PRICE
DELPHI	50.00
JAVA	70.00

BOOKAUTHOR

BOOKNAME	AUTHOR
JAVA	TOM
VC++	JACK

例 3 BOOKPRICE 和 BOOKAUTOR 通过自然连接查询 PRICE 和 AUTHOR。

```
SELECT AUTHOR, PRICE FROM BOOKPRICE NATURAL JOIN BOOKAUTHOR;
```

查询结果如下表 4.2.3 所示。

表 4.2.3 例 3 的查询结果

AUTHOR	PRICE
TOM	70.00

一个自然连接是两个匹配列（在要连接的表中具有相同的列名）的连接。注意：所有可连接的列必须具有相同的列名。

3．笛卡儿积过滤连接Ⅱ：交叉连接(CROSS JOIN)

即笛卡儿积。它和“FROM 表 1，表 2”查询同义。

例 4 BOOKPRICE 和 BOOKAUTOR 通过交叉连接查询 PRICE 和 AUTHOR。

```
SELECT AUTHOR, PRICE FROM BOOKPRICE CROSS JOIN BOOKAUTHOR;
```

查询结果如下表 4.2.4 所示。

表 4.2.4 例 4 的查询结果

AUTHOR	PRICE
TOM	50.0
TOM	70.0
JACK	50.0
JACK	70.0

4．JOIN…ON

这是一种常用的写法。ON 子句引入了一个连接的条件表达式。

例 5 例 1 也可写为：

```
SELECT DISTINCT 商品名,价格
FROM   USER1.厂商登记 A   JOIN   USER1.商品登记 B
ON B.厂商编号=A.厂商编号
WHERE   A.厂商地址='北京';
```

5．JOIN … USING

当连接的两个表中的连接列列名相同时，可以采用 USING 的写法，如下例所示。

例 6 例 1 也可以写为：

```
SELECT DISTINCT 商品名,价格   FROM   厂商登记 JOIN 商品登记
USING (厂商编号)
WHERE   厂商地址='北京';
```

6．自连接

当进行一个基于表内关系的查询时，必须将表与其自身相连接，称为自连接。

例 7 查询电视机价格低于海天电视机厂同类产品的厂商编号，电视机价格,及对方的厂商编号电视机价格。

```
SELECT   T1.厂商编号,T1.价格,T2.厂商编号,T2.价格
FROM   商品登记 T1,商品登记 T2,厂商登记 T3
WHERE   T1.价格<T2.价格 AND   T1.商品名=T2.商品名
AND T2.商品名='电视机' AND T2.厂商编号= T3.厂商编号
AND T3.厂商名='海天电视机厂';
```

查询结果如下表 4.2.5 所示。

表 4.2.5 例 7 的查询结果

T1.厂商编号	T1.价格	T2. 厂商编号	T2. 价格
B0A04	2500.00	B0A04	2800.00

7．内连接(INNER JOIN)

INNER JOIN 可能因为在连接表中没有行与另外一个表中的行相匹配而丢失行。

例 8 从 BOOKPRICE、BOOKAUTHOR 中查询市场上正在出售、有确切作者的书的 PRICE 和 AUTHOR。

```
SELECT AUTHOR, PRICE FROM BOOKPRICE INNER JOIN BOOKAUTHOR
ON BOOKPRICE.BOOKNAME=BOOKAUTHOR.BOOKNAME;
```

查询结果如下表 4.2.6 所示。

表 4.2.6 例 8 的查询结果

AUTHOR	PRICE
TOM	70.00

因为 BOOKAUTHOR 中的 JACK 的书 VC++市场上还没有开始出售，所以这里看不到该记录的信息。如果要想知道该信息，则应该用外部连接查询。

8．外部连接(OUTER JOIN)

一个外部连接将回答“给出一个表 A 和表 B 的连接而不丢失 A 的信息”这样一种形式的问题。

外部连接的类型有 3 种：RIGHT、LEFT 和 FULL。下面分别举例说明。

例 9 从 BOOKPRICE 和 BOOKAUTHOR 中查询所有作者的姓名以及其书的售价：AUTHOR,PRICE。

```
SELECT AUTHOR, PRICE FROM BOOKPRICE RIGHT OUTER JOIN BOOKAUTHOR
ON BOOKPRICE.BOOKNAME=BOOKAUTHOR.BOOKNAME;
```

查询结果如下表 4.2.7 所示。

表 4.2.7 例 9 的查询结果

AUHTOR	PRICE
TOM	70.00
JACK	NULL

例 10 查询市场上所有书的单价及其对应的作者：PRICE,AUTHOR。

```
SELECT AUTHOR, PRICE FROM BOOKPRICE LEFT OUTER JOIN BOOKAUTHOR
ON BOOKPRICE.BOOKNAME=BOOKAUTHOR.BOOKNAME;
```

查询结果如下表 4.2.8 所示。

表 4.2.8 例 10 的查询结果

AUTHOR	PRICE
NULL	50.00
TOM	70.00

例 11 查询所有书的作者与出售单价信息：AUTHOR,PRICE。

```
SELECT AUTHOR, PRICE FROM BOOKPRICE FULL OUTER JOIN BOOKAUTHOR
ON BOOKPRICE.BOOKNAME=BOOKAUTHOR.BOOKNAME;
```

查询结果如下表 4.2.9 所示。

表 4.2.9 例 11 的查询结果

AUTHOR	PRICE
JACK	NULL
TOM	70.00

NULL	50.00

4.3　子查询

在 DM_SQL 语言中，一个 SELECT-FROM-WHERE 语句称为一个查询块，前面讨论的都是只含一个查询块的格式，WHERE 后的条件都是显式的，除谓词所带列名的值不能确定外，其它均是确定值。然而在实际中，经常是给不出确定条件，要先通过对数据库的查询才能确定这些条件，这种包含在查询语句中而又要先做的查询称为子查询。子查询用于规定一个值（标量子查询，返回一个值），或一个表（表子查询，返回一个表）。它通常采用(SELECT…)的形式嵌套在一个表达式内的某个地方。子查询语法如下：

```
<子查询> ::= ( <查询表达式>)
```

即子查询是嵌入括弧的<查询表达式>，而这个<查询表达式>通常是一个 SELECT 语句。它有下列限制：

（1）在子查询中不得有 ORDER BY 子句。

（2）子查询的选择清单中包含 TEXT 和 CLOB 类型的列。允许 TEXT 类型与 CHAR 类型值比较。比较时，取出 TEXT 类型字段的最多 8188 字节与 CHAR 类型字段进行比较；如果比较的两字段都是 TEXT 类型，则最多取 300*1024 字节进行比较。

（3）子查询不应被直接包含在集函数中。

注：在子查询中允许嵌套子查询。

按子查询返回结果的形式，DM 子查询可分为两大类：

（1）标量子查询：只返回一行一列。

（2）表子查询：可返回多行多列。

要说明的是 DM 暂时不支持多列的行子查询：即返回为一行多列的子查询。

4.3.1　标量子查询

依据用户要求和 SQL92 标准，要求支持<标量子查询>作为<值表达式初等项>。<标量子查询>是子查询的一种，其功能是指明从<查询表达式>导出的标量值。

<标量子查询>的度应该是 1，它的数据类型是<标量子查询>中直接包含的<查询表达式>的列的数据类型。如果<标量子查询>的基数大于 1，则产生一个异常：基数违背。简单地说，若一个标量子查询的结果只有一行一列，即单个值，则该值可以参加表达式运算。

下面是一个标量子查询的例子：

```
create table t1( c int, d char(10));
create table t2( c int, d char(20));
insert into t1 values(1, 't1-one');
insert into t2 values(10, 't2-first');
commit work;
```

```
select 'value is', (select c from t1) from t2;
-- t1 只有一列，结果正确
select 'value is', nvl((select c from t1),0) from t2;
-- 函数+标量子查询，结果正确
select 'value is', (select c,d  from t1) from t2;
-- 返回列数不为 1，报错

insert into t1 values(2, 't1-two');
insert into t1 values(3, 't1-three');
insert into t1 values(4, 't1-four');
select count(c) from t1;
select 'value is', (select c from t1) from t2;
-- 子查询返回行值多于一个，报错
delete from t1;
select 'value is', (select c from t1) from t2;
-- t1 有 0 行，结果返回 NULL
rollback work;

update t1 set d = (select d from t2);
update t1 set c = (select avg(c) from t2) + (select avg(c) from t2);
-- Update 语句中允许使用标量子查询

insert into t1(c) values(1+(select c from t2));
-- Insert 语句中允许使用标量子查询
```

4.3.2　表子查询

表子查询经常类似标量子查询，单列构成了子查询的选择清单，但它的查询结果允许返回多列。可以从上下文中区分出表子查询：在其前面始终有一个只对表子查询的算符：<比较算符>ALL、<比较算符>ANY（或是其同义词<比较算符> SOME）、IN、EXISTS 或 UNIQUE。

例 查询提供商品编号为 C0005 的厂商编号、厂商名和厂址。

```
SELECT  厂商编号,厂商名,厂商地址  FROM  厂商登记
WHERE  厂商编号  IN
(  SELECT  厂商编号
   FROM  商品登记  WHERE  商品编号='C0005');
```

查询结果如下表 4.3.1 所示。

表 4.3.1 查询结果

厂商编号	厂商名	厂商地址

B0A05	小鸭洗衣机	上海

该查询语句的求解方式是：首先通过子查询“SELECT 厂商编号 FROM 商品登记 WHERE 商品编号=‘C0005’”在商品登记表中查到商品编号为’C0005’的厂商编号的集合，然后，再到厂商登记表中找到与子查询结果集中的厂商编号所对应的厂商名和厂商地址。

在带有子查询的查询语句中，通常也将子查询称内层查询或下层查询。由于子查询还可以嵌套子查询，相对于下一层的子查询，上层查询又称为父查询或外层查询。

由于 DM_SQL 语言所支持的嵌套查询功能可以将一系列简单查询构造成复杂的查询，从而有效地增强了 DM_SQL 语句的查询功能。以层层嵌套的方式构造语句正是 DM_SQL 的“结构化”的特点。

需要说明的是：上例的外层查询只能用 IN 谓词而不能用比较算符“=”，因为子查询的结果包含多个元组，除非用户能确定子查询的结果只有一个元组时，才能用等号比较。但这对用户来说，有时是很难做到的。建议用户用 IN 谓词，这样总不会出错。如果外层查询要用比较算符与子查询相连，则子查询必须紧跟在比较算符的后面。当然，以上语句也可以用连接查询的方式实现。

```
SELECT   厂商登记.厂商编号,厂商名,厂商地址  FROM  厂商登记,商品登记
WHERE   厂商登记.厂商编号=商品登记.厂商编号  AND  商品编号='C0005';
```

比较这两个语句，读者会发现，后者列名厂商编号前面均加了表名限定符。这是因为两表中有相同列名，为避免混淆必须显式说明，而前者却不用。这是因为在一层子查询中，FROM 后面指出的表名的作用域为该层子查询。因而，在内层子查询中的厂商编号和商品编号应由内层的 FROM 子句后的商品登记表来限定，而外层的厂商编号, 厂商名, 厂商地址均由外层的 FROM 子句后的厂商登记表来限定，由于系统已约定好，用户可以不用表名限定符，如用了也不会出错。

例 查询采购日东冰箱厂所生产商品的商场名和商场地址。

采用子查询嵌套方式写出以下查询语句：

```
SELECT   商场编号,商场名,商场地址   FROM   商场登记
WHERE   商场编号   IN
  (SELECT  商场编号   FROM   供货登记
  WHERE   商品编号   IN
        (SELECT  商品编号   FROM   商品登记
        WHERE   厂商编号  IN
            (SELECT  厂商编号   FROM   厂商登记
            WHERE   厂商名='日东冰箱厂')));
```

查询结果如下表 4.3.2 所示。

表 4.3.2 查询结果

商场编号	商场名	商场地址
A0001	红旗商场	武汉
A0002	上海商场	上海
A0003	王府井商场	北京

该语句采用了四层嵌套查询方式，首先通过最内层子查询从厂商登记表中查询到日

东冰箱厂的厂商编号，再通过第二层子查询从商品登记表中查出该厂商的商品，再通过第三层子查询从供货登记表中查出购买该商品的商场编号，最后由最外层查询，从商场登记表中查询到商场名和商场地址，即得到以上结果。

此例也可用四个表的连接来完成。

从上例可以看出，当查询涉及到多个基表时，嵌套子查询与连接查询相比，前者由于是逐步求解，层次清晰，易于阅读和理解，具有结构化程序设计的优点。但究竟选用哪一种方式用户可根据自己的习惯及运行效率确定。

在许多情况下，外层子查询与内层子查询常常引用同一个表，如下例所示。

例 查询资产总值低于日东电冰箱厂的厂商编号、厂商名、资产总值。

```
SELECT   厂商编号,厂商名,资产总值   FROM   厂商登记
WHERE   资产总值<(SELECT   资产总值   FROM   厂商登记
                    WHERE   厂商名='日东冰箱厂');
```

查询结果如下表 4.3.3 所示。

表 4.3.3 查询结果

厂商编号	厂商名	资产总值
B0A01	华东电视机厂	50.00
B0003	万里鞋厂	30.00
B0A04	海天电视机厂	60.00
B0006	美的服装厂	20.00

此例的子查询与外层查询尽管使用了同一表名，但作用是不一样的。子查询是在该表中查询日东冰箱厂的资产总值，而外查询是在厂商登记表的资产总值列中查找小于该值的集合，从而得到这些值所对应的厂商编号和厂商名。为清楚起见，DM_SQL 语言允许为这样的表引用定义别名：

```
SELECT   厂商编号,厂商名,资产总值   FROM   厂商登记  T1
WHERE   T1.资产总值<(SELECT T2.资产总值
                      FROM   厂商登记  T2
                      WHERE   T2.厂商名='日东冰箱厂');
```

该语句也可以采用连接方式实现：

```
SELECT   T1.厂商编号,T1.厂商名,T1.资产总值
FROM   厂商登记  T1,厂商登记  T2
WHERE   T2.厂商名='日东冰箱厂'   AND   T1.资产总值<T2.资产总值;
```

例　查询供应厂商和仓库都在同一地点的商品名称及其价格。

```
SELECT  商品名,价格
FROM  商品登记  WHERE  仓库地址=ANY
      (SELECT  厂商地址  FROM  厂商登记
      WHERE  厂商编号=商品登记.厂商编号);
```

其结果如下表 4.3.4 所示。

表 4.3.4 查询结果

商品名	价格

电冰箱	2000.00
洗衣机	1000.00

此例有一点需要注意：子查询的 WHERE 子句涉及到商品登记.厂商编号，但是其 FROM 子句中却没有提到商品登记。在外部子查询 FROM 子句中命名了商品登记——这就是外部引用。当一个子查询含有一个外部引用时，它就与外部语句相关联，称这种子查询为**相关子查询**。

例　查询供应厂商和仓库不在同一地点的商品名称及其价格。

```
SELECT 商品名,价格
FROM 商品登记 WHERE 仓库地址<>ALL
        (SELECT 厂商地址 FROM 厂商登记
        WHERE 厂商编号=商品登记.厂商编号);
```

其结果如下表 4.3.5 所示：

表 4.3.5 的查询结果

商品名	价格
电视机	3000.00
鞋_1	50.00
电视机	2500.00
鞋_2	500.00
电视机	2800.00
电冰箱	1900.00

4.3.3　派生表子查询

派生表子查询是一种特殊的表子查询。所谓派生表是指 FROM 子句中的 SELECT 语句，并以别名引用这些派生表。在 SELCET 语句中的 FROM 子句中可以跟随一个或多个派生表。

例 查询存放地址最多的商品名称和地点的数量

```
SELECT 地点数量,商品名 FROM
(   SELECT COUNT(仓库地址), 商品名
    FROM 商品登记 GROUP BY 商品名 )
 AS 商品表 (地点数量, 商品名)
 ORDER BY 地点数量 DESC;
```

查询结果如下表 4.3.6 所示。

表 4.3.6 查询结果

地点数量	商品名
3	电视机
2	电冰箱
1	洗衣机
1	鞋_1

1	鞋_2

4.3.4　定量比较

量化符 ALL、SOME、ANY 可以用于将一个<数据类型>的值和一个由表子查询返回的值的集合进行比较。

1．ALL

ALL 定量比较要求的语法如下：

```
<标量表达式> <比较算符> ALL <表子查询>
```

其中：

（1）<标量表达式>可以是对任意单值计算的表达式。

（2）<比较算符>包括＝、＞、＜、＞＝、＜＝或＜＞。

若表子查询返回 0 行或比较算符对表子查询返回的每一行都为 TRUE，则返回 TRUE。若比较算符对于表子查询返回的至少一行是 FALSE，则 ALL 返回 FALSE。

例 查询未供应商品的厂商编号、厂商名、厂商地址。本例即查询在厂商登记表中有记录而在商品登记表中无相应厂商编号的厂商名和厂商地址。

```
SELECT   DISTINCT   厂商编号,厂商名,厂商地址   FROM   厂商登记
WHERE   厂商编号<>ALL   (SELECT 厂商编号   FROM   商品登记);
```

查询结果如下表 4.3.7 所示。

表 4.3.7 查询结果

厂商编号	厂商名	厂商地址
B0A06	美的服装厂	武汉

例 查询其它厂商供应的、比海天电视机厂所产电视机均贵的电视机的商品编号、价格、厂商编号、仓库地址。

```
SELECT 商品编号,价格,厂商编号,仓库地址   FROM   商品登记
WHERE   价格>ALL
(   SELECT   价格   FROM   商品登记
    WHERE   商品名='电视机'   AND   厂商编号=
    (   SELECT   厂商编号   FROM   厂商登记
        WHERE   厂商名='海天电视机厂'))
AND   商品登记.厂商编号<>ALL
(   SELECT 厂商编号   FROM   厂商登记
    WHERE   厂商名='海天电视机厂')
AND   商品名='电视机';
```

查询结果如下表 4.3.8 所示。

表 4.3.8 查询结果

商品编号	价格	厂商编号	仓库地址
C0001	3000.00	B0A01	武汉

2．ANY 或 SOME

ANY 或 SOME 定量比较要求的语法如下：

```
<标量表达式> <比较算符> ANY | SOME <表子查询>
```

SOME 和 ANY 是同义词。如果它们对于表子查询返回的至少一行为 TRUE，则返回为 TRUE。若表子查询返回 0 行或比较算符对表子查询返回的每一行都为 FALSE，则返回 FALSE。

ANY 和 ALL 与集函数的对应关系如下表所示：

ANY 和 ALL 与集函数的对应关系

	＝	<>	<	<=	>	>=
ANY	IN	不存在	<MAX	<=MAX	>MIN	>=MIN
ALL	不存在	NOT　IN	<MIN	<=MIN	>MAX	>=MAX

在具体使用时，读者完全可根据自己的习惯和需要选用。

4.3.5　带 EXISTS 谓词的子查询

带 EXISTS 谓词的子查询语法如下：

```
<EXISTS 谓词> ::= [NOT] EXISTS <表子查询>
```

EXISTS 判断是对非空集合的测试并返回 TRUE 或 FALSE。若表子查询返回至少一行，则 EXISTS 返回 TRUE，否则返回 FALSE。若表子查询返回 0 行，则 NOT EXISTS 返回 TRUE，否则返回 FALSE。

例　查询武汉各商场所采购商品的编号

```
SELECT   DISTINCT   商品编号   FROM   供货登记  T1
WHERE   EXISTS
(   SELECT    *    FROM    商场登记  T2
    WHERE    T2.商场编号=T1.商场编号    AND    商场地址= '武汉');
```

查询结果如下表 4.3.9 所示。

表 4.3.9　查询结果

商品编号
C0002
C0004
C0005

此例查询需要供货登记表和商场登记表中的数据，其执行方式为：首先在供货登记表的第一行取商场编号的值为 A0001，这样对内层子查询则为：

```
(SELECT    *    FROM    商场登记  T2
WHERE    T2.商场编号='A0001'    AND    商场地址='武汉');
```

在商场登记表中，确实存在满足该条件的行，子查询返回值为真，说明可取供货登记表的第一行作为结果。系统接着取供货登记表的第二行，又得到商场编号的值为 A0001，重复以上步骤……。只有外层子查询 WHERE 子句结果为真时，方可将供货登记表中的对应行送入结果表，如此继续，直到把供货登记表的各行处理完。

通过以上分析，读者就可以领会到，这种子查询与前面介绍的子查询有明显的区别。前面的嵌套子查询是由内向外一层层执行，每层只执行一次，得到一个值或一个集合用于外层查询，内层的查询条件不依赖于外层，这类子查询称为不相关子查询。而本例查询条件依赖于外层查询的某个列，即依赖于供货登记表中的商场编号，并随着它的取值不同而不断变化，反复在执行。它的运行结果也由于内层子查询与外层有关，必须反复求值才能得到，这种查询称相关查询。

在执行时，由于带 EXISTS 谓词的相关子查询只需测试内层查询是否有返回值，给出实际列名也无具体意义，因而其 SELECT 后的值表达式通常都选用*。正因为不需要处理具体值，使它的查询效率有时还高于不相关子查询。

例 查询进了全部商品的商场名。

```
SELECT   商场名   FROM   商场登记
WHERE   NOT   EXISTS
(   SELECT   *   FROM   商品登记
    WHERE   NOT   EXISTS
    (   SELECT   *   FROM   供货登记
        WHERE   商场登记.商场编号=供货登记.商场编号
        AND  商品登记.商品编号=供货登记.商品编号));
```

查询结果为空。

例 查询至少进了 A0003 商场所进全部商品的商场编号。

```
SELECT   DISTINCT   商场编号   FROM   供货登记    A
WHERE   NOT   EXISTS
(   SELECT   *   FROM   供货登记   B
    WHERE   B.商场编号='A0003'   AND   NOT   EXISTS
    (   SELECT   *   FROM   供货登记   C
        WHERE   C.商场编号=A.商场编号
        AND C.商品编号  =B.商品编号));
```

查询结果如下表 4.3.10 所示。

表 4.3.10 查询结果

商场编号
A0001
A0002
A0003

4.4　查询的交

目前 DM 提供了一种集合算符：UNION。这种算符将两个表作为输入并产生一个结果作为输出。集合运算语法如下：

语法格式

```
<查询表达式>
```

```
UNION [ALL]
[ ( ]<查询表达式> [ ) ];
```

使用说明

1．每一个输入的表必须具有相同的列数，而每一个相对应的列对必须具有可比较的数据类型。没有一种情况下允许输入的列含有 BLOB、CLOB 或 IMAGE、TEXT 等多媒体数据类型。

2．在 UNION 后的可选项关键字 ALL 的意思是保持所有重复，而没有 ALL 的情况下表示删除所有重复。

举例说明

表 TABLE1 和 TABLE2 做 UNION，和 UNION ALL 的查询示例。

假设使用两个表：

TABLE1 含有 5 个单列行{0, 1, 2, 2, 3}，TABLE2 含有 5 个单列行{1, 2, 3, 5, 5}。

例 UNION

```
SELECT * FROM TABLE1 UNION SELECT * FROM TABLE2;
```

产生一个来自 TABLE1 和 TABLE2 的所有非重复行，结果为下面 5 行：{0, 1, 2, 3, 5}。

例 UNION ALL

```
SELECT * FROM TABLE1 UNION ALL (SELECT * FROM TABLE2);
```

产生一个来自 TABLE1 和 TABLE2 的所有行：{0, 1, 1, 2, 2, 2, 3, 3, 5, 5}。

4.5 GROUP BY 和 HAVING 子句

4.5.1 GROUP BY 子句的使用

GROUP BY 子句是 SELECT 语句的可选项部分。它定义了成组表。GROUP BY 子句语法如下：

```
<GROUP BY 子句> ::= <列名> | <值表达式>{,<列名> | <值表达式>}
```

GROUP BY 定义了成组表：行组的集合，其中每一个组由其中所有成组列的值都相等的行构成。

例 统计每个厂商所供应的商品种类数。

```
SELECT 厂商编号,COUNT(*) FROM 商品登记 GROUP BY 厂商编号;
```

查询结果如下表 4.5.1 所示。

表 4.5.1 查询结果

厂商编号	COUNT(*)
B0003	2
B0A01	1
B0A02	2

B0A04	2
B0A05	1

系统执行此语句时，首先将商品登记表按厂商编号的取值进行分组，相同的厂商编号为一组，然后对每一组使用集函数 COUNT(*)，统计该组内的记录个数送入结果表，如此继续，直到最后一组，得到查询结果。

如果 GROUP BY 子句前面带有 WHERE 子句时，系统则将不能满足 WHERE 子句条件的行先删去，然后再分组，另外读者要注意 GROUP BY 与 ORDER BY 的区别，前者是按分组列的值将相同列值的行连续地放在一起，组成一个组，但组与组之间并未按分组列值的大小排序，如果要排序，必须再使用 ORDER BY 子句。

例 求地址不在武汉的商场购进商品的总件数，并按商场编号的升序排列。

```
SELECT   A2.商场编号,A1.商场名,SUM(采购商品数量)
FROM   商场登记   A1,供货登记   A2
WHERE   A1.商场编号=A2.商场编号   AND   商场地址<>'武汉'
GROUP   BY   A2.商场编号,A1.商场名
ORDER   BY   A2.商场编号;
```

查询结果如下表 4.5.2 所示。

表 4.5.2 查询结果

商场编号	商场名	SUM(采购商品数量)
A0002	上海商场	1130
A0003	王府井商场	10

使用 GROUP BY 要注意以下问题：

1. 在 GROUP BY 子句中的每一列必须明确地命名属于在 FROM 子句中命名的表的一列。成组列的数据类型不能是多媒体数据类型。

2．成组列不能为集函数表达式或者在 SELECT 子句中定义的别名，但可以是其他表达式。但 SELECT 子句不得包括集函数和成组列以外的列。

3．每个查询语句不管是否带子查询，GROUP BY 子句只能出现一次，不能分组后再分组。

4．当成组列值包含空值时，则空值作为一个独立组。

5．当成组列包含多个列名时，系统则按各列名在 GROUP BY 子句中出现的顺序，首先抽取最左列的列值相同的组成一个个组，在各组内，再按右边的列值相同的组成小组，如此右推，逐个满足，最后得到分组结果。

6．GROUP BY 子句中至多可包含 64 个成组列。

4.5.2　HAVING 子句的使用

HAVING 子句是 SELECT 语句的可选项部分。它也定义了一个成组表。HAVING 子句语法如下：

```
<HAVING  子句> ::= HAVING <搜索条件>
```

HAVING 子句定义了一个成组表，其中只含有搜索条件为 TRUE 的那些组，且通常

跟随一个 GROUP BY 子句。HAVING 子句与组的关系正如 WHERE 子句与表中行的关系。WHERE 子句用于选择表中满足条件的行，而 HAVING 子句用于选择满足条件的组。

例 统计出采购商品总件数大于 100 的商场名、所在地址，并按总件数从小到大的顺序排列。

```
SELECT  商场名,SUM(采购商品数量),商场地址  FROM  供货登记,商场登记
WHERE  供货登记.商场编号=商场登记.商场编号
GROUP  BY 商场名,商场地址
HAVING  SUM(采购商品数量)>100
ORDER  BY  2;
```

查询结果如下表 4.5.3 所示。

表 4.5.3 查询结果

商场名	SUM(采购商品数量)	商场地址
红旗商场	150	武汉
人民商场	200	武汉
上海商场	1130	上海

系统执行此语句时，首先将供货登记表和商场登记表中的各行按相同的商场编号作连接，再按商场名的取值进行分组，相同的商场名为一组，然后对每一组使用集函数 SUM(采购商品数量)，统计该组内采购商品数量的总和，如此继续，直到最后一组。再选择采购商品数量大于 100 的组作为查询结果。

4.6 ORDER BY 子句

ORDER BY 子句可以选择性地出现在<查询表达式>之后，它规定了当行由查询返回时应具有的顺序。ORDER BY 子句的语法如下：

```
<ORDER BY 子句> ::= ORDER BY
    <无符号整数> | <列说明> | <值表达式> [ASC | DESC]
    {,<无符号整数> | <列说明> | <值表达式> [ASC | DESC]}
```

例 将厂商登记表中的资产总值按从大到小的顺序排列。

```
SELECT  *  FROM  厂商登记 ORDER  BY  资产总值  DESC;
```

等价于：

```
SELECT  *  FROM  厂商登记 ORDER  BY  3  DESC;
```

查询结果如下表 4.6.1 所示。

表 4.6.1 查询结果

厂商编号	厂商名	资产总值	厂商地址
B0A05	小鸭洗衣机厂	1000.00	上海
B0A02	日东冰箱厂	80.00	武汉
B0A04	海天电视机厂	60.00	武汉
B0A01	华乐电视机厂	50.00	北京

B0003	万里鞋厂	30.00	北京
B0006	美的服装厂	20.00	武汉

例

```
SELECT  *  FROM  厂商登记  ORDER  BY  5;
```

系统报错：无效的 ORDER BY 语句。

需要说明的是：

1．ORDER BY 子句为 DBMS 提供了要排序的项目清单和他们的排序顺序：递增顺序（ASC，默认）或是递减顺序（DESC）。它必须跟随<查询表达式>，因为它是在查询计算得出的最终结果上进行操作的。

2．排序键可以是任何在查询清单中的列的名称，或者是对最终结果表的列计算的表达式（即使这一列不在选择清单中），但不允许为子查询或集函数。但是对于 UNION 查询语句，排序键必须在第一个查询子句中出现。

3．<无符号整数>为<值表达式>在 SELECT 后的序列号。当用<无符号整数>代替列名时，<无符号整数>不应大于 SELECT 后<值表达式>的个数。如上面例句中 ORDER BY 5，因查询结果列只有 4 列，无法进行排序，系统将会报错。若采用其他常量表达式(如：-1，3×6)作为排序列，将不影响最终结果表的行输出顺序。

4．无论采用何种方式标识想要排序的结果列，它们都不应为多媒体数据类型（如 IMAGE、TEXT、BLOB 和 CLOB）。

5．当排序列值包含 NULL 时，若按升序排列，则含该空值的行排在最后面，若按降序排列则排在最前面。

6．当排序列包含多个列名时，系统则按列名从左到右排列的顺序，先按左边列将查询结果排序，当左边排序列值相等时，再按右边排序列排序……如此右推，逐个检查调整，最后得到排序结果。

7．由于 ORDER BY 只能在最终结果上操作，不能将其放在查询中或一个集函数前。

8．ORDER BY 子句中至多可包含 64 个排序列。

4.7　选取前几条数据

在 DM 中，可以使用“SELECT TOP N”或者“SELECT TOP N PERCENT”语句来筛选 SELECT 结果的前 N 条或是前 N 个百分比的数据。也可以用语句“SELECT TOP N1,N2”语句来选出 SELECT 结果的前 N1 条记录后的 N2 条记录。其中 N、N1、N2 都为正整数。

例　查询价格最贵的两种商品的编号和名称。

```
SELECT TOP 2 商品编号,商品名  FROM  商品登记  ORDER BY  价格  DESC;
```

其结果如表 4.7.1 所示：

表 4.7.1　查询结果

商品编号	商品名
C0001	电视机
C0007	电视机

例　查询前 50%的价格最贵的商品的编号和名称。

```
SELECT TOP 50 PERCENT * FROM 商品登记 ORDER BY 价格 DESC;
```

其结果如表 4.7.2 所示：

表 4.7.2 查询结果

商品编号	商品名	价格	商品照片	仓库地址	厂商编号
C0001	电视机	3000.00	_	武汉	B0A01
C0007	电视机	2800.00	_	北京	B0A04
C0004	电视机	2500.00	_	上海	B0A04
C0002	电冰箱	2000.00	_	武汉	B0A02

例 查询价格第二贵的商品的编号和名称。

```
SELECT TOP 1,1 * FROM 商品登记 ORDER BY 价格 DESC;
```

其结果如表 4.7.3 所示：

表 4.7.3 查询结果

商品编号	商品名	价格	商品照片	仓库地址	厂商编号
C0007	电视机	2800.00	_	北京	B0A04

4.8　全文检索

DM 提供多文本数据检索服务，包括全文索引和全文检索。全文索引为在字符串数据中进行复杂的词搜索提供了有效支持。全文索引存储关于词和词在特定列中的位置信息，全文检索利用这些信息，可快速搜索包含具体某个词或某一组词的记录。

在给定的数据库中执行全文检索涉及到以下这些任务：

对需要进行全文检索的表和列进行注册；

对注册了的列的数据建立全文索引，并用非无关单词填充该索引；

对注册了的列查询填充后的全文索引。

执行全文检索步骤如下：

1．建立全文索引；

2．修改（填充）全文索引；

3．使用带 CONTAINS 谓词的查询语句进行全文检索；

4．当全文索引所在的数据库表数据在全文索引填充之后进行了更新时，可以修改（重新填充）全文索引，以便可以查询更新后的数据；

5．若不再需要，删除全文索引。

6．在全文索引定义并填充后，即可进行全文检索。

全文检索通过在查询语句中使用 CONTAINS 子句进行。CONTAINS 子句格式说明如下：

```
CONTAINS ( <列名> , <检索条件> )
        <检索条件>：：= <布尔项> | <检索条件> <AND | OR> <布尔项>
        <布尔项>：：=[NOT] 布尔因子
        <布尔因子>：：= '字符串'
```

当用户使用 CONTAINS 子句查询时，<列名>必须是已经建立了全文索引并填充后的

列，否则，系统会报错。

有关 DM 全文检索的几点说明：

支持精确字、词、短语及一段文字的查询，CONTAINS 谓词内支持 AND | AND NOT | OR 的使用，AND 的优先级高于 OR 的优先级；

支持对每个精确词（单字节语言中没有空格或标点符号的一个或多个字符）或短语（单字节语言中由空格和可选的标点符号分隔的一个或多个连续的词）的匹配。对词或短语中字符的搜索不区分大小写；

对于短语或一段文字的查询，根据词库，单个查找串被分解为若干个关键词，忽略词库中没有的词和标点符号，在索引上进行（关键词 AND 关键词）匹配查找。因而，不一定是精确查询；

英文查询不考虑大小写和全角半角中英文字符；

不提供 Noise 文件，即不考虑忽略词或干扰词；

不支持通配符“*”；

不提供对模糊词或变形词的查找；

检索条件子句可以和其他子句共同组成 WHERE 的检索条件。

例 全文检索综合实例。

（1）创建一个带有文本列的基表 Employee，并插入 5 条记录。

```
CREATE TABLE Employee(
    NAME            CHAR(20)    PRIMARY KEY,
    AGE             INT,
    ADDRESS         CHAR(100),
    PHONE           CHAR(15),
    RESUME          VARCHAR(300),
    PAPER           TEXT
);
```

```
INSERT INTO Employee VALUES('章章',30, '华工科技产业大厦9楼', '02787522500', '中南财经政法大学，大学本科，工学学士',
'分词就是将连续的字序列按照一定的规范重新组合成词序列的过程。我们知道，在英文的行文中，单词之间是以空格作为自然分界符的，而中文只是字、句和段可以通过明显的分界符来简单划界，唯独词没有一个形式上的分界符. '
)；
```

```
INSERT INTO Employee VALUES('付付',23, '华工科技产业大厦9楼', '02787526641', '华中科技大学，硕士研究生',
'一般来说,全文检索可以使用基于字典的分词,也可以使用单纯的二分词.比如: 美洲豹就可分为: 美洲  州豹.对于英文来说,分词很容易,只需按照空格分词就可以了.比如: Once a day, I bought a hot dog...'
);
```

```
INSERT INTO Employee VALUES('雄雄',25, '华工科技产业大厦9楼', '02787529611', '东北大学，硕士研究生，发表论文一篇',
' 基于字典技术的分词可以减少大量的无意义的分词');
```

```
INSERT INTO Employee VALUES('萧萧',29, '华工科技产业大厦9楼', '02787533500', '东北大学，硕士研究生，发表论文一篇',
' To live or to die, is a question... When he woke up, he saw a black dog watching himself.中文翻译为:活着还是死亡,是一个问题... 当他醒来的时候,一条黑狗正望着他');
```

```
INSERT INTO Employee VALUES('佳佳',24, '华工科技产业大厦9楼', '02787544400', '东北大学，硕士研究生，发表论文一篇',
'Now that you have Maven installed on your system, this section will show you how to integrate it with an existing project.');
```

（2）在 ADDRESS 列和 PAPER 列上分别定义全文索引。

```
CREATE CONTEXT INDEX INDEX1 ON Employee(ADDRESS);
CREATE CONTEXT INDEX INDEX2 ON Employee(PAPER);
```

（3）修改（填充）全文索引。

```
ALTER CONTEXT INDEX INDEX1 ON Employee REBUILD;
ALTER CONTEXT INDEX INDEX2 ON Employee REBUILD;
```

（4）进行全文检索，查找地址中有“华工”字样的雇员的姓名、年龄。

```
SELECT NAME, AGE FROM Employee WHERE CONTAINS(ADDRESS, '华工');
```

结果如表 4.8.1 所示：

表 4.8.1 查询结果

	NAME	AGE
1	章章	30
2	付付	23
3	雄雄	25
4	萧萧	29
5	佳佳	24

（5）进行全文检索，查找地址中有“华工”字样且年龄大于 25 岁的雇员的姓名、年龄。

```
SELECT NAME, AGE FROM Employee WHERE CONTAINS(ADDRESS, '华工') AND AGE > 25;
```

结果如表 4.8.2 所示：

表 4.8.2 查询结果

	NAME	AGE
1	章章	30
2	萧萧	29

注意这里可以将 CONTAINS 子句和其他子句共同组成 WHERE 的检索条件。

（6）进行全文检索，查找论文里有“分词”字样的雇员的姓名、年龄。

```
SELECT NAME, AGE FROM Employee WHERE CONTAINS(PAPER, '分词');
```

结果如表 4.8.3 所示：

表 4.8.3 查询结果

	NAME	AGE
1	章章	30
2	付付	23
3	雄雄	25

(7) 进行全文检索，查找论文里有“分词”及“中文”字样的雇员的姓名、年龄。

```
SELECT NAME, AGE FROM Employee WHERE CONTAINS(PAPER,'分词' AND '中
文');
```

结果如表 4.8.4 所示：

表 4.8.4 查询结果

	NAME	AGE
1	章章	30

（8）进行全文检索，查找论文里有“分词”或“中文”字样的雇员的姓名、年龄。

```
SELECT NAME, AGE FROM Employee WHERE CONTAINS(PAPER,'分词' OR '中文
');
```

结果如表 4.8.5 所示：

表 4.8.5 查询结果

	NAME	AGE
1	章章	30
2	付付	23
3	雄雄	25
4	萧萧	29

（9）进行全文检索，查找论文里无“中文”字样的雇员的姓名、年龄。

```
SELECT NAME, AGE FROM Employee WHERE NOT CONTAINS(PAPER, '中文');
```

结果如表 4.8.6 所示：

表 4.8.6 查询结果

	NAME	AGE
1	付付	23
2	雄雄	25
3	佳佳	24

（10）进行全文检索，查找论文里有“black dog”字样的雇员的姓名、年龄。

```
SELECT NAME, AGE FROM Employee WHERE CONTAINS(PAPER, 'black dog');
```

结果如表 4.8.7 所示：

表 4.8.7 查询结果

	NAME	AGE
1	萧萧	29

（11）进行全文检索，查找论文里有“中文”字样并且地址中有“产业大厦”字样的雇

员的姓名、年龄。

```
SELECT NAME, AGE FROM Employee WHERE CONTAINS(PAPER,
            '中文') AND CONTAINS(ADDRESS, '产业大厦');
```

结果如表 4.8.8 所示：

表 4.8.8 查询结果

	NAME	AGE
1	章章	30
2	萧萧	29

（12）删除论文里有“black dog”字样的雇员记录（即上面第（10）步那条记录），并提交。

```
DELETE   FROM Employee WHERE CONTAINS(PAPER, 'black dog');
COMMIT WORK;
```

（13）修改（重新填充）全文索引，以便可以利用全文索引查询更新后的数据。

```
ALTER CONTEXT INDEX INDEX1 ON Employee REBUILD;
ALTER CONTEXT INDEX INDEX2 ON Employee REBUILD;
```

（14）重复第（11）步的查询，即进行全文检索，查找论文里有“中文”字样并且地址中有“产业大厦”字样的雇员的姓名、年龄。

```
SELECT NAME, AGE FROM Employee WHERE CONTAINS(PAPER,
'中文') AND CONTAINS(ADDRESS, '产业大厦');
```

结果如表 4.8.9 所示：

表 4.8.9 查询结果

	NAME	AGE
1	章章	30

（15）对不再需要的全文索引可以删除。

```
DROP CONTEXT INDEX INDEX1 ON Employee;
DROP CONTEXT INDEX INDEX2 ON Employee;
```

第 5 章　数据的插入、删除和修改

DM_SQL 语言的数据更新语句包括：数据插入、数据修改和数据删除三种语句，其中数据插入和修改二种语句使用的格式要求比较严格。在使用时要求对相应基表的定义，如列的个数、各列的排列顺序、数据类型及关键字、唯一性约束、引用约束、检索约束的内容均要了解得很清楚，否则就很容易出错。下面将分别对这三种语句进行讨论。在讨论中，**如不特别说明，各例的当前数据库均为 CW，用户均为建表者 USER1，因此表名前通常省略了数据库名、模式名前缀。**

5.1　数据插入语句

数据插入语句用于往已定义好的表中插入单个或成批的数据。

INSERT 语句有三种形式。一种形式是值插入，即构造一行，并将它插入到表中；另一种形式为查询插入，它通过计算一个查询以构造要插入表的一行或多行；还有一种形式是过程调用结果插入，即进行过程调用，并将产生的结果集插入到表中。

数据插入语句的语法格式如下：

语法格式

```
INSERT INTO [[<数据库名>.]<模式名>.]<基表或视图名> [(<列名>{,<列名>})]
    VALUES(<插入值>{,<插入值>})
    |<查询说明>
    |<过程调用>;

<基表或视图名>::= <基表名>|<视图名>
```

参数

1．<数据库名>　指明该表或视图所属的数据库，缺省为当前数据库。

2．<模式名>　　指明该表或视图所属的模式，缺省为当前模式。

3．<基表名>　　指明被插入数据的基表的名称。

4．<视图名>　　指明被插入数据的视图的名称，实际上 DM 将数据插入到视图引用的基表中。

5．<列名>　　　表或视图的列的名称。在插入的记录中，这个列表中的每一列都被 VALUES 子句或查询说明赋一个值。如果在此列表中省略了表的一个列名，则 DM 用先前定义好的缺省值插入到这一列中。如果此列表被省略，则在 VALUES 子句和查询中必须为表中的所有列指定值。

6．<插入值>　　指明在列表中对应的列的插入的列值，如果列表被省略了，插入的列值按照基表中列的定义顺序排列。

7．<查询说明>　将一个 SELECT 语句所返回的记录插入表或视图的基表中，子查询中选择的列表必须和 INSERT 语句中列名清单中的列具有相同的数量。查询说明必须遵照的语法规则请参照 SELECT 语句的有关说明。

8．<过程调用>　将过程调用产生的结果集插入到表中，该结果集的列数必须和 INSERT 语句的列名清单中的列数相同。过程调用指调用存储模块，参见 11.3.6 节关于调用语句的说明。

使用说明

1．<基表名>或<视图名>后所跟的<列名>必须是该表中的列，且同一列名不允许出现两次，但排列顺序可以与定义时的顺序不一致。

2．<插入值>的个数、类型和顺序要与<列名>一一对应。

3. 如果某一<列名>未在 INTO 子句后面出现，则新插入的行在这些列上将取空值或缺省值，如该列在基表定义时说明为 NOT NULL 时将会出错。

4．如果<基表名>或<视图名>后没指定任何<列名>，则隐含指定该表或视图的所有列，这时，新插入的行必须在每个列上均有<插入值>。

5. 如果两表之间存在引用和被引用关系时，应先插入被引用表的数据，再插入引用表的数据。

6．<查询说明>是指用查询语句将在另一张表中查得的一组数据插入 INTO 后 <表名>指定的表中，因此该格式的使用可供一次插入多个行。

7．在嵌入方式下工作时，<插入值>可以为主变量。

举例说明

例 1 USER1 用户在厂商登记表中插入一新厂商信息：厂商编号为 B0007，厂商名为幸福公司，资产总值未知，厂商地址在北京。

```
INSERT    INTO    厂商登记(厂商编号,厂商名,厂商地址)
VALUES('B0007', '幸福公司', '北京');
```

该语句运行后，将在厂商登记表中新增加一行。由于资产总值未知，其值为缺省值 10 万，也可以指定为空值, 这时该语句也可以写成以下形式：

```
INSERT    INTO    厂商登记
VALUES('B0007', '幸福公司',NULL, '北京');
```

在定义厂商登记表时，设定了检验约束：CHECK(厂商地址 IN(‘武汉’， ‘北京’，‘上海’))，说明只能是这三个城市的厂商 ，在插入新数据或修改厂商地址时，系统则按检验约束进行检查，如果不满足条件，系统将会报错。

由于 DM 支持标量子查询，标量子查询允许用在标量值合法的地方，因此在数据插入语句的<插入值>位置允许出现标量子查询。

例 2 在商品登记中插入一个新记录，武汉地区等离子电视的价格是普通电视价格的 3 倍。

```
INSERT INTO 商品登记 VALUES(
'C0009',
'等离子电视',
(SELECT (价格) FROM 商品登记 WHERE 商品编号='C0001')*3,
NULL,
'武汉',
'B0A05');
```

若是需要插入一批数据时，可使用带<查询说明>的插入语句。如下例所示：

例 3 USER1 用户构造一新的基表，表名为武汉商品，用来存放武汉生产或存放的商品，列名有：商品编号，商品名，厂商名，价格，关键字为商品编号，并将前面已知表中的数据装入此表中。

```
CREATE TABLE 武汉商品(
   商品编号   CHAR(5)   NOT   NULL,
   商品名     CHAR(20),
   厂商名     CHAR(20),
   价格       NUMERIC(10,2),
   PRIMARY   KEY(商品编号)
);
INSERT INTO 武汉商品
SELECT   DISTINCT   商品编号,商品名,厂商名,价格
FROM   商品登记,厂商登记
WHERE   商品登记.厂商编号=厂商登记.厂商编号
AND   (厂商地址='武汉' OR 仓库地址='武汉');
```

该插入语句的执行将商品登记表和厂商登记表中满足条件的全部数据均装入新建的武汉商品表中。查询结果如下表 5.1.1 所示。

表 5. 1. 1 例 3 的查询结果

商品编号	商品名	厂商名	价格
C0001	电视机	华乐电视机厂	3000．00
C0002	电冰箱	日东冰箱厂	2000．00
C0003	鞋_1	万里鞋厂	50．00
C0004	电视机	海天电视机厂	2500．00
C0006	鞋_2	万里鞋厂	500．00
C0007	电视机	海天电视机厂	2800．00
C0008	电冰箱	日东冰箱厂	1900．00

需要插入一批数据时，还可使用带<过程调用>的插入语句。如下例所示：

例 4 上例的操作也可通过以下语句实现。

```
CREATE TABLE 武汉商品(
   商品编号   CHAR(5)   NOT   NULL,
   商品名     CHAR(20),
   厂商名     CHAR(20),
   价格       NUMERIC(10,2),
   PRIMARY   KEY(商品编号)
);
/* 在执行下面的 CREATE 语句之前，要先创建“商品登记”,“厂商登记”两个表 */
CREATE OR REPLACE PROCEDURE INS_PROC AS
```

```
        BEGIN
            SELECT   DISTINCT   商品编号,商品名,厂商名,价格
            FROM   商品登记,厂商登记
            WHERE   商品登记.厂商编号=厂商登记.厂商编号
            AND   (厂商地址='武汉' OR  仓库地址='武汉');
        END;

    INSERT INTO  武汉商品  CALL INS_PROC;
```

该插入语句的执行将商品登记表和厂商登记表中满足条件的全部数据均装入新建的武汉商品表中。查询结果如下表 5.1.2 所示。

表 5.1.2 例 4 的查询结果

商品编号	商品名	厂商名	价格
C0001	电视机	华乐电视机厂	3000．00
C0002	电冰箱	日东冰箱厂	2000．00
C0003	鞋_1	万里鞋厂	50．00
C0004	电视机	海天电视机厂	2500．00
C0006	鞋_2	万里鞋厂	500．00
C0007	电视机	海天电视机厂	2800．00
C0008	电冰箱	日东冰箱厂	1900．00

值得一提的是，BIT 数据类型值的插入与其他数据类型值的插入略有不同：其他数据类型皆为插入的是什么就是什么，而 BIT 类型取值只能为 1/0，又同时能与整数、精确数值类型、不精确数值类型和字符串类型相容（可以使用这些数据类型对 BIT 类型进行赋值和比较），取值时有一定的规则。

数值类型常量向 BIT 类型插入的规则是：

非 0 数值转换为 1，数值 0 转换为 0。例如：

```
CREATE TABLE T10 (C BIT);
INSERT INTO T10 VALUES(1);        --插入 1
INSERT INTO T10 VALUES(0);        --插入 0
INSERT INTO T10 VALUES(1.2);      --插入 1
```

字符串类型常量向 BIT 类型插入的规则是：

全部由 0 组成的字符串转换为 0，其他全数字字符串（例如'123'），转换为 1。非全数字字符串（例如：'1e1', '2a5', '3.14'）也转换为 1。

```
INSERT INTO T10 VALUES('000');    --插入 0
INSERT INTO T10 VALUES('0');      --插入 0
INSERT INTO T10 VALUES('10');     --插入 1
INSERT INTO T10 VALUES('1.0');    --插入 1
```

5.2　数据修改语句

数据修改语句用于修改表中已存在的数据。

语法格式

```
UPDATE [[<数据库名>.] <模式名>.]<基表或视图名>
        SET <列名>=<值表达式>{,<列名>=<值表达式>}
        [WHERE <条件表达式>];

<基表或视图名>::= <基表名>|<视图名>
```

参数

1．<数据库名>　指明该表或视图所属的数据库，缺省为当前数据库。

2．<模式名>　　指明该表或视图所属的模式，缺省为当前用户的缺省模式。

3．<基表名>　　指明被修改数据的基表的名称。

4．<视图名>　　指明被修改数据的视图的名称，实际上 DM 对视图的基表更新数据。

5．<列名>　　　表或视图中被更新列的名称，如果 SET 子句中省略列的名称，列的值保持不变。

6．<值表达式>　指明赋予相应列的新值。

7．<条件表达式>指明限制被更新的行必须符合指定的条件，如果省略此子句，则修改表或视图中所有的行。

使用说明

1．<值表达式>中不得出现集函数，不得出现子查询，但是可以为 NULL。

2．SET 后的<列名>不得重复出现。

3．WHERE 子句也可以包含子查询，但子查询中出现的表名不可与 UPDATE 后的表名相同。如果省略了 WHERE 子句，则表示要修改所有的元组。

4．如果<列名>为被引用列，只有被引用列中未被引用列引用的数据才能被修改；如果<列名>为引用列，引用列的数据被修改后也必须满足引用完整性。在 DM 系统中，以上引用完整性由系统自动检查。

5．执行基表的 UPDATE 语句触发任何与之相联系的 UPDATE 触发器。

6．如果视图的定义查询中含有以下结构则不能更新视图：

（1）联结运算

（2）集合运算符

（3）GROUP BY 子句

（4）集函数

（5）DISTINCT 运算符

举例说明

例 1 将北京所有厂商的资产总值增加 15%。

```
UPDATE   厂商登记   SET   资产总值=资产总值*(1+0.15)
WHERE   厂商地址='北京';
```

例 2 由于标量子查询允许用在标量值合法的地方，因此在数据修改语句的<值表达式>位置也允许出现标量子查询。下例将武汉地区电视机的价格调节为全国各地电视机的均价。

```
UPDATE 商品登记
SET 价格=(SELECT avg(价格) FROM 商品登记 WHERE 商品名='电视机' and 仓库地址='武汉')
WHERE 商品编号 = 'C0001';
```

注：自增列的修改例外，它一经插入，只要该列存储于数据库中，其值为该列的标识，不允许修改。关于自增列修改的具体情况，请参见 5.5 节——自增列的使用。

5.3　数据删除语句

数据删除语句用于删除表中已存在的数据。

语法格式

```
DELETE  FROM  [[<数据库名>.]<模式名>.]<基表或视图名>
    [WHERE <条件表达式>];

<基表或视图名>::= <基表名>|<视图名>
```

参数

1．<数据库名>　指明该表或视图所属的数据库，缺省为当前数据库。

2．<模式名>　指明该表或视图所属的模式，缺省为当前模式。

3．<基表名>　指明被删除数据的基表的名称。

4．<视图名>　指明被删除数据的视图的名称，实际上 DM 将从视图的基表中删除数据。

5．<条件表达式> 指明基表或视图的基表中被删除的记录须满足的条件。

使用说明

1．如果不带 WHERE 子句，表示删除表中全部元组，但表的定义仍在字典中。因此，DELETE 语句删除的是表中的数据，并未删除表结构。

2．WHERE 子句也可以带子查询，但子查询中出现的表名不可与 DELETE 后面的表名相同。

3．由于 DELETE 语句一次只能对一个表进行删除，因此当两个表存在引用与被引用关系时，要先删除引用表里的记录，只有引用表中无记录时，才能删被引用表中的记录，否则系统会报错。

4．执行与表相关的 DELETE 语句将触发所有定义在表上的 DELETE 触发器。

5．如果视图的定义查询中包含以下结构之一，就不能从视图中删除记录：

（1）联结运算

（2）集合运算符

（3）GROUP BY 子句

（4）集函数

（5）DISTINCT 运算符

例 将没有进货的商场从商场登记表中删去。

```
DELETE   FROM   商场登记
WHERE   商场编号   NOT   IN
    (SELECT   商场编号   FROM   供货登记);
```

由于 DELETE 语句也是一次只能对一个表进行删除，因此当两个表存在引用与被引用关系时，要先删除引用表里的记录，只有引用表中无此记录时，才能删被引用表中的记录，否则系统会报错。

5.4　伪列的使用

5.4.1　ROWID

伪列从语法上和表中的列很相似，查询时能够返回一个值，但实际上在表中并不存在。用户可以对伪列进行查询，但不能插入、更新和删除它们的值。DM 支持的伪列有：ROWID，USER，UID 等。

DM 中行标识符 ROWID 用来标识数据库基表中每一条记录的唯一键值，标识了数据记录的确切的存储位置。如果用户在选择数据的同时从基表中选取 ROWID，在后续的更新语句中，就可以使用 ROWID 来提高性能。如果在查询时加上 FOR UPDATE 语句，该数据行就会被锁住，以防其他用户修改数据，保证查询和更新之间的一致性。例如：

```
SELECT ROWID,厂商编号,厂商名,厂商地址 FROM 厂商登记 WHERE 厂商编号='B0A01';
//假设查询的 ROWID=CF06000000
UPDATE 厂商登记   SET   厂商地址='上海'
WHERE   ROWID=0xCF06000000;
```

5.4.2　UID 和 USER

伪列 USER 和 UID 分别用来表示当前用户的用户名和用户标识。

5.5　DM 自增列的使用

5.5.1　DM 自增列定义

1．自增列功能定义

在表中创建一个标识列。该属性与 CREATE TABLE 语句一起使用。

语法格式

```
IDENTITY [ (种子, 增量) ]
```

参数

1．种子：装载到表中的第一个行所使用的值。

2．增量：增量值，该值被添加到前一个已装载的行的标识值上。增量值可以为正数或负数，但不能为 0。

使用说明

1．IDENTITY 适用于 smallint(-32768～+32767)、int(-2147483648～+2147483647)、bigint(-2^63～+2^63-1) 、decimal(p,0) 或 numeric(p,0)类型的列；对于每个表只能创建一个标识列。

2．不能对标识列使用 DEFAULT 约束。

3．必须同时指定种子和增量值，或者二者都不指定。如果二者都未指定，则取默认值 (1,1)；若种子或增量为小数类型，报错。

4．最大值和最小值为该列的数据类型的边界。

5．标识列一旦生成，无法更新，不允许用 Update 语句进行修改。

2．自增列查询函数

（1） IDENT_SEED（函数）

```
语法：IDENT_SEED ( ‘tablename’ )
```

功能：返回种子值，该值是在带有标识列的表中创建标识列时指定的。

参数：tablename：是带有引号的字符串常量，也可以是变量、函数或列名。tablename 的数据类型为 char 或 varchar。其含义是表名，可带库名、模式名前缀。

返回类型：返回数据类型为 int / NULL

（2）IDENT_INCR（函数）

```
语法：IDENT_INCR ( ‘tablename’ )
```

功能：返回增量值，该值是在带有标识列的表中创建标识列时指定的。

参数：tablename：是带有引号的字符串常量，也可以是变量、函数或列名。tablename 的数据类型为 char 或 varchar。其含义是表名，可带库名、模式名前缀。

返回类型：返回数据类型为 int / NULL

例 用自增列查询函数获得自增列的种子和增量信息。

```
CREATE TABLE TIND (
    IND             INT IDENTITY(2,10),
    CONTENT         VARCHAR(50)
);
INSERT INTO TIND VALUES('identity table');
SELECT IDENT_SEED('TIND') FROM TIND;
查询结果为：2
SELECT IDENT_INCR('TIND') FROM TIND;
```

查询结果为：10

5.5.2 SET IDENTITY_INSERT 属性

设置是否允许将显式值插入表的标识列中。

语法格式

```
SET IDENTITY_INSERT [[<数据库名>.]<模式名>.]<表名> ON | OFF;
```

参数

1．<数据库名> 指明表所属的数据库，缺省为当前数据库。

2．<模式名> 指明表所属的模式，缺省为当前模式。

3．<表名> 指明含有标识列的表名。

使用说明

1．IDENTITY_INSERT 属性的默认值为 OFF。SET IDENTITY_INSERT 的设置是在执行或运行时进行的。当一个连接结束，IDENTITY_INSERT 属性将被自动还原为 OFF。

2．DM 要求一个连接中只有一个表的 IDENTITY_INSERT 属性可以设置为 ON。当一个表的 IDENTITY_INSERT 属性被设置为 ON 时，该表中的自动增量列的值由用户指定。如果插入值大于表的当前标识值（自增列当前值），则 DM 自动将新插入值作为当前标识值使用，即改变该表的自增列当前值；否则，将不影响该自增列当前值。

3．当设置一个表的 IDENTITY_INSERT 属性为 OFF 时，新插入行中自增列的当前值由系统自动生成，用户将无法指定。

4．自增列一经插入，无法修改。

举例说明

例：将 IDENTITY 属性与 CREATE TABLE 一起使用

（1）下面的示例创建一个新表，该表将 IDENTITY 属性用于获得自动增加的标识号。

```
CREATE TABLE employees
(
 id_num int IDENTITY(1,1),
 fname varchar (20),
 minit char(1),
 lname varchar(30)
);
```

（2）在该表中插入数据，自增列的值由系统自动生成。

```
INSERT into employees (fname, minit, lname) VALUES ('Karin', 'F', 'Josephs');
INSERT into employees (fname, minit, lname) VALUES ('Pirkko', 'O', 'Koskitalo');
```

插入结果如表 5.5.1 所示：

表 5.5.1 插入结果

id_num	fname	Minit	Lname
1	Karin	F	Josephs

2	Pirkko	O	Koskitalo

（3）仅当表名后有显式的列名列表，并且 IDENTITY_INSERT 为 ON 时，才能在表‘employees’中为标识列指定显式值。例如：

```
SET IDENTITY_INSERT employees ON;
INSERT into employees values(5, 'aa', 'a', 'aaa');
INSERT into employees(id_num) values (5);
INSERT into employees(id_num, fname, minit, lname) values (6, 'aa', 'a', 'aaa');
```

插入结果如表 5.5.2 所示：

表 5. 5. 2 插入结果

id_num	Fname	Minit	Lname
1	Karin	F	Josephs
2	Pirkko	O	Koskitalo
5	aa	a	aaa
5	<NULL>	<NULL>	<NULL>
6	aa	a	aaa

（4）不允许用户修改自增列的值。

```
update employees set id_num=3 where id_num=5;
```

修改失败。对于自增列，不允许 UPDATE 操作。

（5）还原 IDENTITY_INSERT 属性。

```
SET IDENTITY_INSERT employees OFF;
```

（6）插入后再次查询。注意观察自增列当前值的变化。

```
INSERT into employees(fname, minit, lname) VALUES ('End', 'E', 'Bye');
```

表 5.5.3 查询结果

id_num	fname	minit	Lname
1	Karin	F	Josephs
2	Pirkko	O	Koskitalo
5	aa	a	aaa
5	<NULL>	<NULL>	<NULL>
6	aa	a	aaa
7	End	E	Bye

第 6 章　视图

视图是从一个或几个基表(或视图)导出的表，它是一个虚表，即数据字典中只存放视图的定义(由视图名和查询语句组成)，而不存放对应的数据，这些数据仍存放在原来的基表中。当需要使用视图时，则执行其对应的查询语句，所导出的结果即为视图的数据。因此当基表中的数据发生变化时，从视图中查询出的数据也随之改变了，视图就象一个窗口，透过它可以看到数据库中用户感兴趣的数据和变化。由此可见，视图是关系数据库系统提供给用户以多种角度观察数据库中数据的重要机制，体现了数据库本质最重要的特色和功能，它简化了用户数据模型，提供了逻辑数据独立性，实现了数据共享和数据的安全保密。视图是数据库技术中一个十分重要的功能。

视图一经定义，就可以和基表一样被查询、修改和删除，也可以在视图之上再建新视图。由于对视图数据的更新均要落实到基表上，因而操作起来有一些限制，读者应注意如何才能在视图中正确更新数据。在本章各例中，**如不特别说明，各例的当前数据库均为 CW，用户均为建表者 USER1，因此表名、视图名前通常省略了数据库名、模式名前缀。**

6.1　视图的定义

语法格式

```
CREATE    [OR REPALCE]    VIEW
      [<模式名>.]<视图名>[(<列名> {,<列名>})]
      AS <查询说明>
      [WITH    CHECK    OPTION];
```

参数

1．<模式名>　　指明被创建的视图属于哪个模式，缺省为当前模式。

2．<视图名>　　指明被创建的视图的名称。

3．<列名>　　　指明被创建的视图中列的名称。

4．<查询说明>　标识视图所基于的表的行和列。其语法遵照 SELECT 语句的语法规则。

5．WITH CHECK OPTION　指明往该视图中插入或修改数据时，插入行或更新行的数据必须满足视图定义中<查询说明>所指定的条件，如果不带该选项，则插入行或更新行的数据不必满足视图定义中<查询说明>所指定的条件。

语句功能

供用户定义视图。

使用说明

1．<视图名>后所带<列名>不得同名，个数必须与<查询说明>中 SELECT 后的<值表达式>的个数相等。如果<视图名>后不带<列名>，则隐含该视图中的列由<查询说明>中 SELECT 后的各<值表达式>组成，但这些<值表达式>必须是单纯列名。如果出现以下三种情况之一，<视图名>后的<列名>不能省：

（1）<查询说明>中 SELECT 后的<值表达式>不是单纯的列名，而包含集函数或运算表达式。

（2）<查询说明>包含了多表连接，使得 SELECT 后出现了几个不同表中的同名列作为视图列。

（3）需要在视图中为某列取与<查询说明>中 SELECT 后<列名>不同的名字。

最后要强调的是：<视图名>后的<列名>只能全部省或全部不省，没有中间选择。

2．为了防止用户通过视图更新基表数据时，无意或故意更新了不属于视图范围内的基表数据，在视图定义语句的子查询后提供了可选项 WITH CHECK OPTION，如选择，表示往该视图中插入或修改数据时，要保证插入行或更新行的数据满足视图定义中<查询说明>所指定的条件，不选则可不满足。

3．如果视图查询中包含下列结构：连接、集合运算符、GROUP BY 子句，则在视图上不能进行插入、修改和删除操作。

4．视图是一个逻辑表，它自己不包含任何数据。视图的作用可参见本章 6.5 节。

权限

该语句的使用者必须对<查询说明>中的每个表均具有 SELECT 权限。

举例说明

例 1 假定当前数据库为 CW，用户 USER1 构造一视图，名为北京厂商登记，用来存放北京厂商的基本情况，列名有：厂商编号，厂商名，资产总值，厂商地址。

```
CREATE  VIEW  北京厂商登记  AS
        SELECT  厂商编号,厂商名,资产总值,厂商地址
        FROM  厂商登记  WHERE  厂商地址='北京'
        WITH  CHECK  OPTION;
```

由于视图列名与查询说明中 SELECT 后的列名相同，所以视图名后的列名可省。

运行该语句，AS 后的查询语句并未执行，系统只是将所定义的<视图名>及<查询说明>送数据字典保存，对用户来说，好象在数据库中已经有北京厂商登记这样一个表。

如果对该视图作查询：

```
SELECT  *  FROM  北京厂商登记;
```

查询结果见下表 6.1.1。

表 6.1.1 例 1 的查询结果

厂商编号	厂商名	资产总值	厂商地址
B0A01	华乐电视机厂	50	北京
B0003	万里鞋厂	30	北京

用户可以在该表上作数据库的查询、插入、删除、修改等操作。在建好的视图之上还可以再建立视图。

由于以上定义包含可选项 WITH CHECK OPTION，以后对该视图作插入、修改和删除操作时，系统均会自动用 WHERE 后的条件作检查，不满足条件的数据，则不能通过该视图更新相应基表中的数据。

例 2 视图也可以建立在多个基表之上。构造一视图，名为武汉商品_VIEW，用来存放武汉生产或存放的商品，列名有：商品编号、商品名、厂商名、价格。

```
CREATE  VIEW  武汉商品_VIEW  AS
```

```
        SELECT  商品编号,商品名,厂商名,价格
        FROM    商品登记,厂商登记
        WHERE   商品登记.厂商编号=厂商登记.厂商编号
        AND   (厂商地址='武汉'   OR   仓库地址='武汉');
```

如果对该视图作查询：

```
    SELECT   *   FROM   武汉商品_VIEW;
```

查询结果见下表 6.1.2。

表 6. 1. 2 例 2 的查询结果

商品编号	商品名	厂商名	价格
C0001	电视机	华乐电视机厂	3000.00
C0002	电冰箱	日东冰箱厂	2000.00
C0003	鞋_1	万里鞋厂	50.00
C0004	电视机	海天电视机厂	2500.00
C0006	鞋_2	万里鞋厂	500.00
C0007	电视机	海天电视机厂	2800.00
C0008	电冰箱	日东冰箱厂	1900.00

由前面的介绍可知，基表中的数据均是基本数据。为了减少数据允余，由基本数据经各种计算统计出的数据一般是不存贮的，但这样的数据往往又要经常使用，这时可将它们定义成视图中的数据。

例 3 在供货关系上建立一视图，用于统计商场购进某一商品（即商品编号相同）的总件数。

```
CREATE   VIEW   采购商品统计(商场编号,商品编号,商品总件数)   AS
        SELECT   商场编号,商品编号,SUM(采购商品数量)
        FROM   供货登记   GROUP   BY   商场编号,商品编号;
```

在该语句中，由于 SELECT 后出现了集函数 SUM(采购商品数量)，不属于单纯的列名，所以视图中的对应列必须重新命名，即在<视图名>后明确说明视图的各个列名。

由于该语句中使用了 GROUP BY 子句，所定义的视图也称分组视图。分组视图的<视图名>后所带<列名>不得包含集函数。

如果对该视图作查询：

```
    SELECT   *   FROM   采购商品统计;
```

查询结果如下表 6.1.3 所示。

表 6.1.3 查询结果

商场编号	商品编号	商品总件数
A0001	C0002	100
A0001	C0004	50
A0002	C0002	130
A0002	C0003	1000

A0003	C0002	10
A0004	C0005	200

6.2　视图的删除

一个视图本质上是基于其他基表或视图上的查询，我们把这种对象间关系称为依赖。用户在创建视图成功后，系统还隐式地建立了相应对象间的依赖关系。在一般情况下，当一个视图不被其他对象依赖时可以随时删除视图。

语法格式

```
DROP VIEW [<模式名>.]<视图名> [RESTRICT | CASCADE];
```

参数

1．<模式名>　　指明被删除视图所属的模式，缺省为当前模式。

2．<视图名>　　指明被删除视图的名称。

3．CASCADE　　强制删除该视图。

4．RESTRICT　　若视图上有依赖对象，无法删除该视图。

使用说明

1．视图删除有两种方式：RESTRICT/CASCADE 方式。其中 RESTRICT 为缺省值。如果在该视图上建有其它视图，必须先删除参考此视图的视图或者使用 CASCADE 参数强制删除该视图，否则删除视图的操作不会成功。

2．使用 CASCADE 参数删除视图时，如果存在参考该视图的视图，那么 DM 会先删除那个参考视图，再删除本视图；如果没有删除参考视图的权限，那么两个视图都不会被删除。

3. 该视图删除后，用户在其上的权限也均自动取消，以后系统中再建的同名视图，是与它毫无关系的视图。

权限

使用者必须拥有 DBA 权限或是该视图的建立者。

举例说明

例 1 删除视图北京厂商登记，可使用下面的语句：

```
DROP    VIEW    CW.USER1.北京厂商登记;
```

当该视图对象被其他对象依赖时，用户在删除视图时必须带 CASCADE 参数，系统会将依赖于该视图的其他数据库对象一并删除，以保证数据库的完整性。

例 2 USER1 用户删除视图北京厂商登记，同时删除此视图上的其他视图，可使用下面的语句：

```
DROP    VIEW    北京厂商登记  CASCADE;
```

6.3　视图的查询

视图一旦定义成功，对基表的所有查询操作都可用于视图。对于用户来说，视图和

基表在进行查询操作时没有区别。

例 查询武汉电视机的商品编号,厂商名,价格。

```
SELECT　商品编号,厂商名,价格　FROM　武汉商品_VIEW
WHERE　商品名='电视机';
```

系统执行该语句时，先从数据字典中取出视图武汉商品_VIEW 的定义，按定义语句查询基表，得到视图表，再根据条件:商品名=‘电视机’查询视图表, 选择所需列名,得到结果如下表 6.3.1 所示。

表 6.3.1 查询结果

商品编号	厂商名	价格
C0001	华乐电视机厂	3000．00
C0004	海天电视机	2500．00
C0007	海天电视机	2800．00

视图尽管是虚表，但它仍可与其它基表或视图作连接查询，也可以出现在子查询中。

例 统计商场采购某一武汉商品的总件数。

```
SELECT　A.商场编号,商场名,商品名,商品总件数,商场地址
FROM　商场登记　A, 武汉商品_VIEW　B, 采购商品统计　C
WHERE　A.商场编号=C.商场编号　AND　B.商品编号=C.商品编号;
```

系统执行该语句时,先从数据字典中取出视图武汉商品_VIEW 和采购商品统计的定义，按定义语句查询基表，得到两个视图表，再将商场登记表和两个视图表按连接条件作连接，选择所需列名,得到最后结果。

得到结果如下表 6.3.2 所示。

表 6.3.2 查询结果

A.商品编号	商场名	商品名	商品总件数	商场地址
A0001	红旗商场	电冰箱	100	武汉
A0002	上海商场	电冰箱	130	上海
A0003	王府井商场	电冰箱	10	北京
A0002	上海商场	鞋_1	1000	上海
A0001	红旗商场	电视机	50	武汉

6.4　视图数据的更新

视图数据的更新包括插入(INSERT)、删除(DELETE)和修改(UPDATE)三类操作。由于视图是虚表，并没有实际存放数据，因此对视图的更新操作均要转换成对基表的操作。在 SQL 语言中，对视图数据的更新语句与对基表数据的更新语句在格式与功能方面是一致的。

例 1 将北京厂商的资产总值增加 10%。

由于前面已建立了视图北京厂商登记，此修改操作可直接在该视图上进行：

```
UPDATE　北京厂商登记　SET　资产总值=资产总值*(1+0.1);
```

系统执行该语句时，首先从数据字典中取出视图北京厂商登记的定义，将其中的查

询说明与对视图的修改语句结合起来，转换成对基表的修改语句，然后再执行这个转换后的更新语句：

```
UPDATE  厂商登记  SET  资产总值=资产总值*(1+0.1)
WHERE  厂商地址='北京';
```

例 2 往视图北京厂商登记中插入一个新的记录，其中厂商编号为 B0007，厂商名为大地鞋厂，资产总值为 500 万元，厂商地址为上海。则相应的插入语句为：

```
INSERT  INTO  北京厂商登记  VALUES('B0007', '大地鞋厂',500, '上海');
```

对于视图北京厂商登记来说，该语句是一错误语句。根据定义该视图的条件和选项 WITH CHECK OPTION 子句，要求插入数据必须满足条件：厂商地址=‘北京’，而待插入行的厂商地址为上海，所以该行不能通过视图进入基表。如果定义该视图时不选择 WITH CHECK OPTION 子句，该语句可以执行。

例 3 删除北京厂商登记中资产总值低于 35 万的厂商记录。

```
DELETE  FROM  北京厂商登记  WHERE  资产总值<35;
```

系统将该语句与北京厂商登记视图的定义相结合，转换成对基表的语句：

```
DELETE  FROM  厂商登记  WHERE  厂商地址='北京'  AND  资产总值
<35;
```

系统执行该语句，完成厂商登记表的删除。

但应注意的是：北京厂商登记表尽管是视图，在做更新时一样要考虑基表间的引用关系。因为厂商登记表与商品登记表存在着引用关系，商品登记表为引用表，厂商登记为被引用表，只有当引用表中没有相应厂商编号时才能删除厂商登记表中相应记录。

在关系数据库中，并不是所有视图都是可更新的，即并不是所有的视图更新语句均能有意义地转换成相应的基表更新语句，有些甚至是根本不能转换。例如视图厂商统计的定义为：

```
CREATE  VIEW  厂商统计(厂商地址, 厂商总数)  AS
SELECT  厂商地址,  COUNT(*)  FROM  厂商登记
GROUP BY  厂商地址;
```

如果要将武汉的厂商总数改为 100，相应的修改语句为：

```
UPDATE  厂商统计  SET  厂商总数=100  WHERE  厂商地址='武汉';
```

由于厂商总数是查询结果按厂商地址分组后各组所包含的行数，这是无法修改的。象这样的视图为不可更新视图。

目前，不同的关系数据库管理系统产品对更新视图的可操作程度均有差异。DM 系统有这样的规定：

（1）如果视图由两个以上的基表导出时，则该视图不允许更新。

（2）如果视图建在单个基表或单个可更新视图上，且该视图包含了表中的全部 PRIMARY KEY，则该视图为可更新视图。

（3）如果视图列是集函数，或视图定义中的查询说明带了 GROUP BY 子句或 HAVING 子句，则该视图不允许更新。

（4）在不允许更新视图之上建立的视图也不允许更新。

应该说明的是：只有当视图是可更新的时，才可以选择 WITH CHECK OPTION 项。

6.5　视图的作用

视图是提供给用户以多种角度观察数据库中数据的重要机制。尽管在对视图作查询和更新时有各种限制，但只要用户对 DM_SQL 语言熟悉，合理使用视图对用户建立自己的管理信息系统会带来很多的好处和方便，归纳起来，主要有以下几点：

1．用户能通过不同的视图以多种角度观察同一数据

例如，前面建立的商品采购管理信息系统，对不同的用户来说，所需观察的数据是有差别的。商场采购员最关心本地商品的价格、质量、商场进货总的情况，对厂商的情况并不感兴趣，而基表中这些数据并未直接存放，需要通过统计计算后才能得到。这时则可针对不同需要建立相应视图，使他们从不同的需要来观察同一数据库中的数据。

2．简化了用户操作

由于视图是从用户的实际需要中抽取出来的虚表，因而从用户角度来观察这种数据库结构必然简单清晰，另外，由于复杂的条件查询已在视图定义中一次给定，用户再对该视图查询时也简单方便得多了。

3．为需要隐蔽的数据提供了自动安全保护

所谓“隐蔽的数据”是指通过某视图不可见的数据。由于对不同用户可定义不同的视图，使需要隐蔽的数据不出现在不应该看到这些数据的用户视图上，从而由视图机制自动提供了对机密数据的安全保密功能。

4．为重构数据库提供了一定程度的逻辑独立性

在建立调试和维护管理信息系统的过程中，由于用户需求的变化、信息量的增长等原因，经常会出现数据库的结构发生变化，如增加新的基表，或在已建好的基表中增加新的列，或需要将一个基表分解成两个子表等，这称为数据库重构。数据的逻辑独立性是指当数据库重构时，对现有用户和用户程序不产生任何影响。

在管理信息系统运行过程中，重构数据库最典型的示例是将一个基表垂直分割成多个表。将经常要访问的列放在速度快的服务器上，而不经常访问的列放在较慢的服务器上。例如将商品登记(商品编号，商品名，价格，商品照片，仓库地址，厂商编号), 分解为两个基表：

商品登记_1　(商品编号，商品名，价格，仓库地址)

商品登记_2　(商品编号，商品照片，厂商编号)

并将商品登记表中的数据分别插入这两个新建表中，再删去商品登记表。这样一来，原有用户程序中有关商品登记表的操作就均无法进行了。为了减少对用户程序影响，这时可在商品登记_1 和商品登记_2 两基表上建立一个名字为商品登记的视图,因为新建视图维护了用户外模式的原状，用户的应用程序不用修改仍可通过视图查询到数据，从而较好支持了数据的逻辑独立性。

第 7 章　嵌入式 SQL

DM_SQL 尽管功能强，使用方便灵活，但由于它本身没有过程性结构，大多数语言都是独立执行、与上下文无关，很难满足大型管理信息系统的需要。为了解决这一问题，DM_SQL 语言提供了两种工作方式：交互方式和嵌入方式。这样一来，既发挥了高级语言数据类型丰富、处理方便灵活的优势，又以 DM_SQL 语言弥补了高级语言难以描述数据库操作的不足，从而为用户提供了建立大型管理信息系统和处理复杂事务所需要的工作环境。在这种方式下使用的 DM_SQL 语言称为 DM 嵌入式 SQL，而 DM 嵌入式 SQL 的高级语言称为主语言或宿主语言。

嵌入在主语言程序中的 DM_SQL 语句并不能直接被主语言编译程序所识别，必须对这些 SQL 语句进行预处理，将其翻译成主语言语句，生成由主语言语句组成的目标文件，然后再由编译程序编译成可执行文件。为了能够识别嵌入在主语言中的 SQL 语句，必须按一定的规则来编写嵌入式 SQL 代码：

1．以对主语言合适的 SQL 前缀开始每一个 SQL 语句。

2．以对主语言合适的 SQL 终结符结束每一个 SQL 语句。

3．说明所有的将在一些特殊 DECLARE 段与 SQL 共享的主语言程序变量。

4．说明用于错误处理的附加程序变量。

在本章各例中，**如不特别说明，各例的当前数据库均为 CW，用户均为建表者 USER1，因此表名、视图名前通常省略了数据库名、模式名前缀。**

7.1　SQL 前缀和终结符

嵌入式 SQL 语句必须具有一个前缀和一个终结符。DM 嵌入式 SQL 语句的前缀是 EXEC SQL，而终结符是分号。在 EXEC SQL 和分号之间必须是有效的 SQL，不能有主语言语句及其注释。DM 嵌入式 SQL 前缀语法如下：

语法格式

```
EXEC SQL <SQL 语句>;
```

参数

<SQL 语句>　指明使用的嵌入式 DM_SQL 语句。

使用说明

1．该语句只能在嵌入式方式中使用。EXEC 和 SQL 必须在一起出现，不能在分开的行中。

2．DM 嵌入式 SQL 语句包括说明性嵌入式 SQL 语句和可执行嵌入式 SQL 语句两类。说明性语句包括：嵌入变量声明节的开始和结束语句、异常声明语句和游标声明语句。除此之外，其它的数据操纵语句皆为可执行的 SQL 语句，可执行语句又分为数据定义、数据控制、数据操纵三种。

3．除特别声明的，在嵌入方式下有不同语法的语句(如 SELECT 语句)以外，所有的在 DM_SQL 文本中定义的语句都可以作为 DM 嵌入式 SQL 语句。

7.2　宿主变量

嵌入式 SQL 可以在任何可放置标量表达式的地方含有宿主语言变量。宿主变量可用来在程序和 SQL 数据之间传送数据。在 SQL 语句中，<宿主变量名>必须前置一个冒号以同 SQL 对象名相区别。宿主变量只能返回标量值。

声明节是任何嵌入式 SQL 程序的一个重要部分。变量说明出现在 EXEC SQL BEGIN DECLARE SECTION 和 EXEC SQL END DECLARE SECTION 之间。声明节语法如下：

语法格式

```
EXEC SQL BEGIN DECLARE SECTION;
        {<宿主变量定义语句>;}
EXEC SQL END DECLARE;

<宿主变量定义语句> ::= <宿主变量名> <宿主变量数据类型>
```

参数

1．<宿主变量名>　　　指明在 DM 嵌入式 SQL 中使用的宿主变量的名称。

2．<宿主变量数据类型> 指明在 DM 嵌入式 SQL 中使用的宿主变量的数据类型。

使用说明

1．该语句只能在嵌入式方式中使用。

2．在声明节中定义的变量可以被 C 语句使用，而不必要在声明节之外重新定义这些变量，否则会引起变量重复定义的错误。

3．在 DM_SQL 中所使用的宿主变量均必须在 DM 嵌入式 SQL 声明节中加以说明。

4．一个程序模块中可以出现多个声明节，各声明节可出现在 C 程序的说明语句可出现的位置上。PROC*C 规定，一个模块中所有的声明节中的宿主变量不得同名，而不论其实质是全局变量还是局部变量。预编译时，相应声明节的 C 语句是去掉了“EXECSQL BEGIN DECLARE SECTION”和“EXEC SQL ENDDECLARE SECTION”两个语句后的变量定义语句，其位置顺序均保持不变。

举例说明

例 定义宿主变量。

```
EXEC SQL BEGIN DECLARE SECTION;
      INT      number;
      CHAR     string[20];
EXEC SQL END DECLARE SECTION;
```

该例中 number 和 string 是宿主变量。它们根据宿主语言（在此是 C）的规定定义。预编译器要求知道每个宿主变量的数据类型、大小和名称。

7.2.1　输入和输出变量

宿主变量分为输入宿主变量和输出宿主变量两种。输入和输出都是从 DBMS 的角度而言的，“输入”指输入到 DBMS，而“输出”指从 DBMS 输出。很容易找到嵌入式 SQL

程序中的宿主变量，因为它们的名称前面都带有冒号。语法格式为：

```
: <宿主变量>
```

如下例所示：

```
EXEC SQL CREATE TABLE T1 ( C1 INT, C2 CHAR(20) );
number = 1;
strcpy(string, “1234”);
EXEC SQL INSERT INTO T1 VALUES ( :number, :string );
```

注意：宿主变量的数据类型必须与 SQL 语句中的列类型相容。

7.2.2　指示符变量

为了能够处理 NULL 值，在主语言中引入了指示符变量。它是一个跟在宿主变量后的数字变量，标记该宿主变量的数据是否为空。在 DM 嵌入式 SQL 语句中，指示符语法如下：

```
: <宿主变量> [:<指示符名>]
```

如下例所示：

```
EXEC SQL BEGIN DECLAR SECTION
… … /*其他宿主变量的定义*/
short str_indicator;
EXEC SQL BEGIN DECLAR SECTION
… …
str_indicator = -1;
EXEC SQL INSERT INTO T1 VALUES ( :number, :string :str_indicator);
/*  插入结果：对 T1 的 C2 列插入一个 NULL 值  */
```

对于输入宿主变量，预置指示符变量的值为-1，表明插入一个空值。

对于输出宿主变量，指示符表示返回结果是否为空或有截取的情况。一个指示变量的取值有三种：取值为 0，说明返回值非空且赋给宿主变量时不发生截取；取值为-1，说明返回的值为空值；取值>0，说明返回的值为非空值，但在赋给宿主变量时，字符串的值被截断，这时指示变量的值为截断之前的长度。

7.3　服务器登录与退出

在 DM 嵌入方式下，用户与 DBMS 进行交互前，必须先用命令 LOGIN 登录 DM 服务器。在与服务器交互结束后，用 LOGOUT 命令与服务器断开连接。

7.3.1　登录服务器

嵌入方式下，供用户向系统报告自己的登录名和口令，以便系统检查是否为合法用

户。

语法格式

```
LOGIN <嵌入式变量 1> PASSWORD <嵌入式变量 2> SERVER <嵌入式变量 3>;
```

参数

1．<嵌入式变量 1>　指明登录系统的用户使用的登录的名称，该用户当前须具有数据库级权限。

2．<嵌入式变量 2>　指明与登录系统的登录名对应的口令。

3．<嵌入式变量 3>　指明所登录系统服务器的主机名。

使用说明

1．该语句只能在嵌入式方式中使用。

2．<嵌入式变量 1>、<嵌入式变量 2>和<嵌入式变量 3>必须是在嵌入程序说明节中说明过的字符串变量。

3．嵌入式变量 1、嵌入式变量 2 为长度不超过 129 的字符串指针(在 DM 中登录名和密码最多为 128 个字符)。

4．在用户程序中，登录语句的逻辑位置必须先于所有可执行的 SQL 语句，只有执行登录语句之后，才能执行其它可执行的 SQL 语句。

7.3.2　退出服务器

嵌入方式下，用户用来断开与数据库的连接。

语法格式

```
LOGOUT;
```

参数

无

使用说明

1．该语句只能在嵌入式方式中使用。

2．在程序结束时，需要执行 LOGOUT 语句，以断开与服务器的连接，回收服务器的资源。

举例说明

例 在 PRO * C 嵌入式环境中，用户 SYSDBA 以 SYSDBA 登录名登录到服务器 DMSERVER 上：

```
#include <stdio.h>
#include <string.h>
EXEC SQL BEGIN DECLARE SECTION;
   char LOGINNAME[20] = "SYSDBA";
   char USERPWD[20] = "SYSDBA";
   char APPSRV[20] = "DMSERVER";
EXEC SQL END DECLARE SECTION;
long    SQLCODE;
```

```
void main( )
{
    EXEC SQL LOGIN : LOGINNAME PASSWORD :USERPWD SERVER :APPSRV;
    if (SQLCODE != 0)
    {
      printf("用户%s 登录失败\n\n", LOGINNAME);
      return ;
    }
    else
      printf("用户%s 登录成功\n\n", LOGINNAME);
    EXEC SQL LOGOUT;
    printf("用户%s 断开成功\n\n", LOGINNAME);
}
```

7.4　游标的定义与操纵

由于 DM_SQL 语言是面向集合的语言，一条 SQL 语句可以产生或处理多条记录，而高级语言是面向记录的，一组主变量一次只能存放一条记录，所以仅使用主变量并不能满足 SQL 语句向应用程序输出数据的要求。为解决这一问题，SQL 语言引用了游标来协调这两种不同的工作方式,从而为应用程序逐个处理查询结果提供了强有力的手段。DM_SQL 语言提供了四条有关游标的语句：定义游标语句、打开游标语句、拨动游标语句和关闭游标语句。

使用游标必须先定义，定义游标实际上是定义了一个游标工作区，并给该工作区分配了一个指定名字的指针（即游标）。游标工作区用以存放满足查询条件行的集合，因此它是一说明性语句，是不可执行的。在打开游标时，就可从指定的基表中取出所有满足查询条件的行送入游标工作区并根据需要分组排序，同时将游标置于第一行的前面以备读出该工作区中的数据。当对行集合操作结束后，应关闭游标，释放与游标有关的资源。下面分述这几个语句的使用格式及使用中的注意事项。

7.4.1　定义游标语句

定义一个游标，给它一个名称与之相联系的 SELECT 语句。

语法格式

```
DECLARE <游标名> CURSOR FOR <查询说明>;
```

参数

1．<游标名>　　指明被定义的游标的名称。

2．<查询说明>　指明对应于被定义的游标的 SELECT 语句。

使用说明

1．该语句只能在嵌入方式或过程中使用。

2．<查询说明>的语法请参考 SELECT 语句的语法，并且该 SELECT 语句不能包括 INTO 子句。

3．用户必须在其他的嵌入式 SQL 语句引用该游标之前定义它。游标说明的范围在整个预编译单元内，并且每一个游标的名字必须在此范围内唯一。

4．在嵌入方式中，通过游标对基表进行修改和删除时要求该游标表必须是可更新的。即游标定义中给出的查询说明必须是可更新的。DM 系统对可更新的查询说明规定如下：

（1）查询说明的 FROM 后只带一个表名，且该表必须是基表或者是可更新视图。

（2）查询说明是在一个基表的行列子集上，SELECT 后的每个值表达式均由单一的列名组成，且该基表的主关键字均出现在值表达式中。

（3）查询说明不能带 GROUP BY 子句、HAVING 子句或 DISTINCT、ORDER BY 子句。

不满足以上条件的游标表是不可更新的，可更新游标的条件与可更新视图的条件是一致的。

举例说明

例 定义一游标，游标名为 WHSP，用来查询武汉生产或存放的商品，选择的列名有：商品编号、商品名、厂商名、价格，按商品编号的升序排列。

```
DECLARE WHSP CURSOR FOR
    SELECT   商品编号, 商品名, 厂商名，价格
    FROM   商品登记, 厂商登记
    WHERE   商品登记.厂商编号=厂商登记.厂商编号
    AND (厂商地址='武汉'   OR   仓库地址='武汉')
    ORDER BY  商品编号;
```

该语句定义了名字为 WHSP 的游标，该游标要从商品登记和厂商登记表中导出满足以下条件的行：厂商地址在武汉或者仓库地址在武汉的商品编号、商品名、厂商名及价格，并按商品编号排序（升序）。定义了存取这个行集合的工作区及取数指针 WHSP。

7.4.2　打开游标语句

打开一已经定义过的游标。

语法格式

```
OPEN <游标名>;
```

参数

<游标名>　　指明被打开的游标的名称。

使用说明

1．打开游标必须是打开已定义过的游标。

2．该语句的执行将根据定义游标语句中的查询说明所指出的条件，从表中取出符合条件的行进入游标工作区，然后将游标置于第一行之前的位置。

举例说明

例 打开 7.4.1 节例中定义的游标 WHSP：

```
OPEN    WHSP;
```

系统执行该语句，则将查询结果送入游标工作区，并将游标置于第一行之前，这时游标工作区的状态如下表 7.4.1 所示。

WHSP→　　　　表 7.4.1 WHSP 游标工作区

商场编号	商品名	厂商名	价格
C0001	电视机	华乐电视机厂	3000.00
C0002	电冰箱	日东冰箱厂	2000.00
C0003	鞋_1	万里鞋厂	50.00
C0004	电视机	海天电视机厂	2500.00
C0006	鞋_2	万里鞋厂	500.00
C0007	电视机	海天电视机厂	2800.00
C0008	电冰箱	日东冰箱厂	1900.00

7.4.3　拨动游标语句

使游标移动到指定的一行，若游标名后跟随有 INTO 子句，则将游标当前指示行的内容取出分别送入 INTO 后的各变量中。

语法格式

```
FETCH [NEXT|PRIOR|FIRST|LAST|ABSOLUTE n|RELATIVE n] <游标名>
                  [INTO <主变量名>{,<主变量名>}];
```

参数

1．NEXT　　　游标下移一行。
2．PRIOR　　　游标前移一行。
3．FIRST　　　游标移动到第一行。
4．LAST　　　游标移动到最后一行。
5．ABSOLUTE　游标移动到第 n 行。
6．RELATIVE　游标移动到当前指示行后的第 n 行。
7．<游标名>　　指明被拨动的游标的名称。
8．<主变量名>　指明存储数据的变量名。

使用说明

1．待拨动的游标必须是已打开过的游标。

2．DM 通过增加的[NEXT|PRIOR|FIRST|LAST|ABSOLUTE n|RELATIVE n]属性设置，可以方便地将游标拨动到结果集任意位置获得数据。游标拨动的默认属性为 NEXT。但这些增强属性只能在过程中使用。

3．当游标被打开后，若不指定游标的移动位置，第一次执行 FETCH 语句时，游标下移，指向工作区中的第一行，以后每执行一次 FETCH 语句，游标均顺序下移一行，使这一行成为当前行。

4．INTO 后的变量个数、类型必须与定义游标语句中、SETECT 后各值表达式的个

数、类型一一对应。

5．在 PROC*C 中进行游标查询时，当返回值的数据类型与宿主变量的数据类型不一致时，DM 系统将返回值转换成宿主变量的类型。这种转换只局限于数值转换。不论数据类型如何，如果返回给宿主变量的值是 NULL，那么相应指示符变量被置为-1。如果没有与之相应的指示符变量，那么 SQLCODE 被设置为-5000。数值型数据转换，包括 short、int、double、float 四种数值型数据的转换，若发生溢出错误，将给出警告信息。

6．如果当前游标已经指向查询的最后一条记录，在 PROC*C 中使用 FETCH 语句将会导致返回错误代码(SQLCODE=100)；若是在存储过程中使用，将返回一个状态码告知用户已经到达最后一条记录。

举例说明

例 1 对于 7.4.2 节中刚打开的游标，当执行以下语句两次后，INTO 后各变量的值为多少（设 INTO 后的主变量与定义游标语句中 SELECT 后的值表达式在个数，类型上均一一对应）。

```
FETCH    WHSP    INTO    A, B, C, D;
```

由于是两次连续执行 FETCH 语句且结果均放在变量 A、B、C、D 中，这样，第一次的结果已被第二次的结果冲掉，最后的结果为：

A=‘C0002’；B=‘电冰箱’；C=‘日东冰箱厂’；D=2000.00

该例说明：当需要连续取出工作区的多行数据时，应将 FETCH 语句置入高级语言的循环结构中。

例 2 当设置 ABSOLUTE 属性时，n 值是从 1 开始的。如在 7.4.2 节中的游标，执行如下语句：

```
FETCH ABSOLUTE 3 WHSP INTO A, B, C, D;
```

结果为：

A=‘C0003’; B=‘鞋_1’; C=‘万里鞋厂’;　D=50.00

如果执行：

```
FETCH ABSOLUTE 0 WHSP INTO A, B, C, D;
```

则游标的 NOTFOUND 将被置为 TRUE。

7.4.4　关闭游标语句

关闭指定的游标，收回它所占的资源。

语法格式

```
CLOSE   <游标名>;
```

参数

<游标名>　指明被关闭的游标的名称。

使用说明

1．该语句只能在嵌入方式或过程中使用。

2．待关闭的游标必须是已打开过的游标。

3．游标一旦关闭，就不能再对该游标使用 FETCH 语句，否则应重新打开游标。

举例说明

例 要关闭前面已打开的 WHSP 游标，应使用以下语句：

```
CLOSE    WHSP;
```

7.4.5　关于可更新游标

在嵌入方式或过程中，通过游标对基表进行修改和删除时要求该游标表必须是可更新的。可更新游标的条件是：

1．游标定义中给出的查询说明必须是可更新的。DM 系统对查询说明是可更新的有这样的规定：

(1) 查询说明的 FROM 后只带一个表名，且该表必须是基表或者是可更新视图。

(2) 查询说明是在一个基表的行列子集上，SELECT 后的每个值表达式均由单一的列名组成，且该基表的主关键字、唯一性列及未定义缺省值的全部 NOT NULL 列也均出现在值表达式中。

(3) 查询说明不能带 GROUP BY 子句、HAVING 子句或 DISTINCT。

(4) 查询说明不能嵌套子查询。

2．游标定义中不能带 ORDER BY 子句。

不满足以上条件的游标表是不可更新的，例如，前面 7.4.1 节例子中所定义的游标 WHSP 就是不可更新的。其实，可更新游标的条件与可更新视图的条件是一致的。

例 定义一游标 TV，用来查询商品电视机的基本情况，选择的列名有：商品编号、价格、仓库地址。

```
DECLARE TV CURSOR FOR
        SELECT    商品编号,价格,仓库地址    FROM    商品登记
        WHERE     商品名='电视机';
```

显而易见，TV 是可更新游标。

而 7.4.1 节例子中定义的游标 WHSP 是一个不可更新游标。

7.4.6　游标定位删除语句

DM 系统除了提供一般的数据删除语句外，还提供了游标定位删除语句。

语法格式

```
DELETE FROM [[<数据库名>.]<模式名>.] <基表或视图名>
     [WHERE CURRENT OF <游标名>];
  <基表或视图名>::=<基表名> | <视图名>
```

参数

1．<数据库名>　指明游标所对应表或视图所属数据库名，缺省为当前数据库名。

2．<模式名>　指明游标所对应表或视图所属模式名，缺省为当前模式名。

3．<基表名>　指明游标所对应表名。

4．<视图名>　指明游标所对应视图名。

5．<游标名>　　指明使用的游标的名称。

使用说明

1．语句中的游标在程序里已定义并被打开。

2．指定的游标表应是可更新的。

3．该基表应是游标定义中第一个 FROM 子句中所标识的表。

4．游标结果集必须确定，否则 WHERE CURRENT OF <游标名>无法定位。

举例说明

例　已知游标的定义如 7.4.5 节定义的游标所示，当前库为 CW，USER1 用户的程序中有以下语句，请分析执行结果。

```
OPEN    TV;
FETCH    ABSOLUTE    2    TV;
DELETE    FROM    商品登记    WHERE    CURRENT    OF    TV;
```

USER1 用户是建表者，拥有该表上的所有权限。游标打开后，指针指在游标表的第一行的前面，游标工作区的状态如下表 7.4.2 所示。

TV→　　表 7.4.2 TV 游标工作区

商品编号	价格	仓库地址
C0001	3000．00	武汉
C0004	2500．00	上海
C0007	2800．00	北京

执行 FETCH 语句后，使游标下移两行，再执行 DELETE 语句，删除了指针所指的第二行，游标顺序下移，最后的结果如下表所示(说明：因商品登记表中的商品编号为被引用列，此例应假设供货登记表中已无商品编号为 C0004 的行)。

表 7. 4. 3

商品编号	价格	仓库地址
C0001	3000．00	武汉
C0007	2800．00	北京

→

7. 4. 7　游标定位修改语句

DM 系统除了提供一般的数据修改语句外，还提供了游标定位修改语句。

语法格式

```
UPDATE [[<数据库名>.]<模式名>.] <基表或视图名>
        SET <列名>=<值表达式>{,<列名>=<值表达式>}
        [WHERE CURRENT OF <游标名>];
<基表或视图名>::=<基表名> | <视图名>
```

参数

1．<数据库名>　指明游标所对应表或视图所属数据库名，缺省为当前数据库名。

2．<模式名>　　指明游标所对应表或视图所属模式名，缺省为当前模式名。

3．<基表名>　　指明游标所对应表名。

4．<视图名>　　指明游标所对应视图名。

5．<游标名>　　指明游标的名称。

6．<列名>　　　指明表或视图中被更新列的名称，如果 SET 子句中省略列的名称，列的值保持不变。

7．<值表达式>　指明赋予相应列的新值。

使用说明

1．语句中的游标在程序里应已被定义并打开。

2．指定的游标表应是可更新的。

3．该表应是游标定义中第一个 FROM 子句中所标识的表，所指的<列名>必须是表中的一个列，且不应在语句中多次出现。

4．语句中的值表达式不应包含集函数说明。

5．如果指定的表是可更新视图，其视图定义中使用了 WITH CHECK OPTION 子句，则该语句所给定的列值不应产生使视图定义中 WHERE 条件为假的行。

6．游标结果集必须确定，否则 WHERE CURRENT OF <游标名>无法定位。

举例说明

例　已知游标的定义如 7.4.5 节定义的游标，当前库为 CW，USER1 用户的程序中有以下语句，请分析执行结果。

```
OPEN    TV;
FETCH ABSOLUTE 3 TV;
UPDATE   商品登记   SET  价格=1300    WHERE    CURRENT    OF    TV;
```

USER1 用户是建表者，拥有该表上的所有权限。游标打开后，指针指在游标表的第一行的前面，再执行 FETCH 语句，游标下移三行，再执行 UPDATE 语句，将游标所指的第三行中的价格修改为 1300，最后的结果如下表 7.4.4 所示。

表 7.4.4

	商品编号	价格	仓库地址
	C0001	3000．00	武汉
	C0004	2500．00	上海
→	C0007	2800．00	北京

7.5　单元组查询语句

游标为应用程序逐个处理 DM_SQL 语言产生的多条查询结果提供了强有力的手段，但当用户明知查询结果唯一时（如主关键字查询，查询结果或唯一或不存在），仍使用游标处理会比较麻烦。为方便用户操作，DM_SQL 语言提供了单元组查询语句：从指定表中查询满足条件的一行，并将取到的数据赋给对应的变量。它的语法格式如下：

语法格式

```
SELECT [ALL | DISTINCT] <值表达式> {, <值表达式>}
```

```
        INTO <主变量名>{, <主变量名>}
        FROM   [[<数据库名>.]<模式名>.] <基表或视图名>|<相关名>
               {,[[<数据库名>.]<模式名>.] <基表或视图名>|<相关名>}
        [WHERE <搜索条件>]
    <基表或视图名>::=<基表名> | <视图名>
```

参数

1．ALL　　　　返回所有被选择的行，包括所有重复的拷贝，缺省值为 ALL。

2．DISTINCT　　从被选择出的具有重复行的每一组中仅返回一个这些行的拷贝。

3．<值表达式>　　从在 FROM 子句中列出的表、视图中选择一表达式，通常是基于列的值。如果表、视图具有指定的用户名限定，则列要用用户名限定。

4．<主变量名>　　指明存储数据的变量名。

使用说明

1．用户对该语句中包含的每个<表名>应具有 SELECT 特权。

2．该语句中不应包含<GROUP BY 子句>或<HAVING 子句>，且不应出现分组视图。

3．INTO 后的主变量个数、类型必须与 SETECT 后<值表达式>的个数、类型一一对应。

4．WHERE 子句中的查询条件不得带集函数。

5．不处理多媒体数据类型。

6．如果没有行满足 WHERE 子句条件，则没有行能被获取并且 DM 返回一错误码。

举例说明

例 1 分析以下语句的执行结果：

```
        EXEC SQL BEGIN DECLARE SECTION;
            char A[10];
            int B;
            char C[20];

        EXEC SQL END DECLARE SECTION;

        EXEC SQL SELECT   商品名，价格，厂商名
            INTO      :A,:B,:C
            FROM      商品登记,厂商登记
            WHERE     商品编号='C0005'
            AND  商品登记.厂商编号=厂商登记.厂商编号;
```

查询结果为：

A=‘洗衣机’；　B=1000；　C=‘小鸭洗衣机厂’

例 2 分析以下语句的执行结果：

```
        EXEC SQL BEGIN DECLARE SECTION;
            ……
            int B;
        EXEC SQL END DECLARE SECTION;
```

```
……
EXEC SQL SELECT   SUM(采购商品数量)
     INTO      :B
     FROM      供货登记
     WHERE     商场编号='A0001';
```

查询结果为：

B=150。

7.6　动态 SQL

静态 SQL 语句提供的编程灵活性在许多情况下仍显得不足，有时候需要编写更为通用的程序，如当查询条件是不确定的，或者要查询的属性列也是不确定的，就无法用一条静态 SQL 语句实现了。因为在这些情况下，SQL 语句不能完整地写出来，而且这类语句在每次执行时都还有可能变化，只有在程序执行时才能构造完整。象这种在程序执行过程中临时生成的 SQL 语句叫动态 SQL 语句。

实际上，如果在预编译时下列信息不能确定，就必须使用动态 SQL 技术：

（1）SQL 语句正文；

（2）主变量个数；

（3）主变量的数据类型；

SQL 语句中引用的数据库对象（例如：列、索引、基本表、视图等）。

动态 SQL 方法允许在程序运行过程中临时“组装”SQL 语句，主要有三种形式：

（1）语句可变；

（2）条件可变；

（3）数据对象、查询条件均可变。

这几种动态形式几乎覆盖所有的可变要求。为了实现上述三种可变形式，在 DM 嵌入式 SQL 中提供了相应的语句：EXECUTE IMMEDIATE、PREPARE、EXECUTE。下面分别介绍这三种语句。

7.6.1　EXECUTE IMMEDIATE 立即执行语句

用立即执行语句执行动态 SQL 语句。

语法格式

```
EXECUTE IMMEDIATE <SQL 动态语句文本>;
```

参数

<SQL 动态语句文本>　　指明立即执行的动态语句文本。

功能

动态地准备和执行一条语句。

使用说明

1．该语句只能在嵌入式方式中使用。

2．该语句首先分析动态语句文本，检查是否有错误，如果有错误则不执行它，并在 SQLCODE 中返回错误码；如果没发现错误则执行它。

3．用该方法处理的 SQL 语句一定不是 SELECT 语句，而且不包含任何虚拟的输入宿主变量。

4．一般来说，应该使用一个字符串变量来表示一个动态 SQL 语句的文本，下列语句不能作动态 SQL 语句：CLOSE、DECLARE、FETCH、OPEN、WHENEVER。

5．如果动态 SQL 语句的文本中有多条 SQL 语句，那么只执行第一条语句。

6．对于仅执行一次的动态语句，用立即执行语句较合适。

举例说明

例 假定变量 SQLSTMT 是声明节中定义的字符数组：

```
strcpy(SQLSTMT,"DELETE FROM  商品登记;");
EXEC SQL EXECUTE IMMEDIATE :SQLSTMT;
```

7.6.2　PREPARE 准备语句

嵌入方式下，为 EXECUTE 语句的执行准备一条语句。

语法格式

```
PREPARE <SQL 语句名> FROM <SQL 动态语句文本>;
```

参数

1．<SQL 语句名>　　　　指明被分析的动态语句文本的标识符。

2．<SQL 动态语句文本>　指明被分析的动态语句文本。

使用

1．该语句只能在嵌入式方式中使用。

2．<SQL 语句名>标识被分析的动态 SQL 语句，它是供预编译程序使用的标识符，而不是宿主变量。

3．SQL 动态语句文本中的 SQL 语句，不能是 SELECT 语句。

4．该语句可能包含虚拟输入宿主变量（用问号表示），而且变量的类型是已知的。

5．如果 SQL 动态语句文本中含有多条 SQL 语句，那么只执行第一条 SQL 语句(第一条以分号结束的 SQL 语句)。

7.6.3　EXCUTE 执行语句

嵌入方式下，执行一个由 PREPARE 准备好的动态 SQL 语句。

语法格式

```
EXECUTE <SQL 语句名> [<结果使用子句>];

<结果使用子句> ::= USING :<实宿主变量名>{,:<实宿主变量名>}
```

参数

1．<SQL 语句名>　　指明准备执行的动态语句文本的标识符。

2．<结果使用子句>　指出一个实宿主变量表，用于替换虚宿主变量。

3．<实宿主变量名>　指明用于替换虚宿主变量的相应的实宿主变量的名称。

使用说明

1．该语句只能在嵌入式方式中使用。

2．该语句执行前，必须先使用 PREPARE 语句准备一个 SQL 语句并获得 SQL 语句名。

3．<结果使用子句>中的实宿主变量要与被分析的动态 SQL 语句中的虚宿主变量在类型、次序上相对应，个数相匹配。

举例说明

例 假定变量 shop_no 和 SQLSTMT 是声明节中已定义的字符数组，下面 4 条语句的功能相当于执行删除语句“DELETE FROM 商场登记 WHERE 商场编号= ‘B0007 ’;”

```
strcpy(shop_no, "B0007 ");
strcpy(SQLSTMT, "DELETE FROM 商场登记 WHERE 商场编号=? ");
EXEC SQL PREPARE STMT FROM :SQLSTMT;
EXEC SQL EXECUTE STMT USING :shop_no;
```

7.7　异常处理

在每个嵌入式 SQL 语句后，都可以加上一段代码来询问 DBMS 是否执行正确，DBMS 会返回 SQLSTATE 诊断信息码来告知用户是否出错及错误的类别。WHENEVER 语句可以用来说明在该异常申明处理语句的作用域内每个 SQL 操纵语句在发生操作异常时的处理方法。它的格式如下：

语法格式

```
WHENEVER SQLERROR | NOT FOUND
    CONTINUE |
    GOTO <标号> |
    GO TO <标号> |
    DO <宿主语句>;
```

参数

1．SQLERROR　指明当返回码 SQLCODE<0 即语句执行错误时，进行异常处理。

2．NOT FOUND　指明当返回码 SQLCODE=100 即查询语句没有结果返回时，进行异常处理。

3．CONTINUE　指明即使产生异常，也不需要作异常处理。其主要作用是取消前面与之具有相同条件的异常申明处理语句的作用。

4．GOTO 子句　指明程序转移到标号处继续执行。GOTO 可以书写为 GO TO。

5．<标号>　指明跳转的位置，它为标准的 C 语言的标号。

使用说明

1．该语句只能在嵌入式方式中使用。

2．一个 WHENEVER 语句的作用域是从该语句出现的位置开始，到下一个指明相

同条件的 WHENEVER 语句出现（若没有下一个相同条件的 WHENEVER 语句，则到文件结束）之前的所有 SQL 语句。这种起始、终止位置是指源程序的物理位置，与该程序逻辑执行顺序无关。

举例说明

例　下面举例说明 WHENEVER 命令在 PROC*C 嵌入式 SQL 程序中的用法。

```
        EXEC SQL WHENEVER SQLERROR GOTO sql_error;
        ……
sql_error:
      EXEC SQL WHENEVER SQLERROR CONTINUE;
```

第 8 章　函数

在值表达式中，除了可以使用常量、列名、集函数等之外，还可以使用函数作为组成成份。DM 中支持的函数分为数值函数、字符串函数、日期时间函数、空值判断函数、类型转换函数和杂类函数等。本手册还给出了 DM 部分系统函数的详细介绍。下列各表给出了函数的简要说明。在本章各例中，**如不特别说明，各例的当前数据库均为 CW，用户均为建表者 USER1，因此表名、视图名前通常省略了数据库名、模式名前缀。**

数值函数

序号	函数名	功能简要说明
01	ABS(n)	求数值 n 的绝对值
02	ACOS(n)	求数值 n 的反余弦值
03	ASIN(n)	求数值 n 的反正弦值
04	ATAN(n)	求数值 n 的反正切值
05	ATAN2(n1,n2)	求数值 n1/n2 的反正切值
06	CEIL(n)	求大于或等于数值 n 的最小整数
07	CEILING(n)	求大于或等于数值 n 的最小整数，等价于 CEIL
08	COS(n)	求数值 n 的余弦值
09	COT(n)	求数值 n 的余切值
10	COSH(n)	求数值 n 的双曲余弦值
11	DEGREES(n)	求弧度 n 对应的角度值
12	EXP(n)	求数值 n 的自然指数
13	FLOOR(n)	求小于或等于数值 n 的最大整数
14	LN(n)	求数值 n 的自然对数
15	LOG(n1[,n2])	求数值 n2 以 n1 为底数的对数
16	LOG10(n)	求数值 n 以 10 为底的对数
17	MOD(m,n)	求数值 m 被数值 n 除的余数
18	PI()	得到常数 π
19	POWER(n1,n2)	求数值 n2 以 n1 为基数的指数
20	RADIANS(n)	求角度 n 对应的弧度值
21	RAND([n])	求一个 0 到 1 之间的随机浮点数
22	ROUND(n[,m])	求四舍五入值函数
23	SIGN(n)	判断数值的数学符号
24	SIN(n)	求数值 n 的正弦值
25	SINH(n)	求数值 n 的双曲正弦值
26	SQRT(n)	求数值 n 的平方根
27	TAN(n)	求数值 n 的正切值

28	TANH(n)	求数值 n 的双曲正切值
29	TRUNC(n[,m])	截取数值函数
30	TRUNCATE(n[,m])	截取数值函数，等价于 TRUNC

字符串函数

序号	函数名	功能简要说明
01	ASCII(char)	求字符的 ASCII 码
02	BIT_LENGTH(char)	求字符串的位长度
03	CHAR(n)	将 ASCII 码转换成为字符
04	CHAR_LENGTH(char)/ CHARACTER_LENGTH(char)	求字符串的串长度
05	CHR(n)	将 ASCII 码值转换成为字符,等价于 CHAR
06	CONCAT(char1,char2)	顺序联结两个字符串成为一个字符串
07	DIFFERENCE(char1,char2)	以整数返回两个字符串的 SOUNDEX 值之差
08	INITCAP(char)	将字符串中单词的首字符转换成大写的字符
09	INSERT(char1,n1,n2,char2) / INSSTR(char1,n1,n2,char2)	将字符串 char1 从 n1 的位置开始删除 n2 个字符，并将 char2 插入到 char1 中 n1 的位置
10	INSTR(char1,char2[n,[m]])	从输入字符串 char1 的第 n1 个字符开始查找字符串 char2 的第 n2 次的出现，以字符计算
11	INSTRB(char1,char2[n,[m]])	从输入字符串 char1 的第 n1 个字符开始查找字符串 char2 的第 n2 次的出现，以字节计算
12	LCASE(char)	将大写的字符串转换为小写的字符串
13	LEFT(char,n) / LEFTSTR(char,n)	返回字符串最左边的 n 个字符组成的字符串
14	LENGTH(char)	返回字符串的长度，尾部空格不计数
15	LENGTHB(char)	返回输入字符串的字节数
	OCTET_LENGTH(char)	返回字符串的字节数,等价于 LENGTHB
16	LOCATE(char1,char2[,n])	返回 char1 在 char2 中首次出现的位置
17	LOWER(char)	将大写的字符串转换为小写的字符串
18	LPAD(char1,n,char2)	在输入字符串的左边填充上 char2 指定的字符，将其拉伸至 n 个字符长
19	LTRIM(char1,char2)	从输入字符串中删除所有的前导字符，

		这些前导字符由 char2 来定义
20	POSITION(char1,/ IN char2)	求串 1 在串 2 中第一次出现的位置
21	REPEAT(char,n) / REPEATSTR(char,n)	返回将字符串重复 n 次形成的字符串
22	REPLACE(char,search_string[,replacem e nt_string])	将输入字符串中所有出现的 search_s tring 都替换成 replace_string 字符串
23	REVERSE(char)	将字符串反序
24	RIGHT / RIGHTSTR(char,n)	返回字符串最右边 n 个字符组成的字符串
25	RPAD(char1,n,char2)	类似 LPAD 函数，只是向右拉伸该字符串使之达到 n 个字符串长
26	RTRIM(char1,char2)	从输入字符串的右端开始删除 char2 参数中的字符
27	SOUNDEX(char)	返回一个表示字符串发音的字符串
28	SPACE(n)	返回一个包含 n 个空格的字符串
29	SUBSTR(char,m,n) / SUBSTRING(char FROM m [FOR n])	从输入字符串中取出一个子串，从 m 字符处开始取指定长度的字符串
30	SUBSTRB(char,n,m)	SUBSTR 函数等价的单字节形式
31	TO_CHAR(DATE[,fmt])	日期转换为字符串函数
32	TRANSLATE(char,from,to)	将所有出现在搜索字符集中的字符转换成字符集中的相应字符
33	TRIM([LEADING|TRAILING|BOTH] [char1] FROM char2)	删去字符串 char2 中由串 char1 指定的字符
34	UCASE(char)	将小写的字符串转换为大写的字符串
35	UPPER(char)	将小写的字符串转换为大写的字符串

日期时间函数

序号	函数名	功能简要说明
01	ADD_DAYS(date,n)	返回日期加上 n 天后的新日期
02	ADD_MONTHS(date,n)	在输入日期上加上指定的几个月返回一个新日期
03	ADD_WEEKS(date,n)	返回日期加上 n 个星期后的新日期
04	CURDATE()	返回系统当前日期
05	CURTIME()	返回系统当前时间
06	CURRENT_DATE()	返回系统当前日期
07	CURRENT_TIME(n)	返回系统当前时间
08	CURRENT_TIMESTAMP(n)	返回系统当前时间戳
09	DATEADD(datepart,n,date)	向指定的日期加上一段时间
10	DATEDIFF(datepart,date1,date2)	返回跨两个指定日期的日期和时间边界数
11	DATEPART(datepart,date)	返回代表日期的指定部分的整数

12	DAYNAME(date)	返回日期的星期名称
13	DAYOFMONTH(date)	返回日期为所在月份中的第几天
14	DAYOFWEEK(date)	返回日期为所在星期中的第几天
15	DAYOFYEAR(date)	返回日期为所在年中的第几天
16	DAYS_BETWEEN(date1,date2)	返回两个日期之间的天数
17	EXTRACT(时 间 字 段 FROM date)	抽取日期时间或时间间隔类型中某一个字段的值
18	GETDATE()	返回系统当前时间戳
19	HOUR(time)	返回时间中的小时分量
20	LAST_DAY(date)	返回输入日期所在月份最后一天的日期
21	MINUTE(time)	返回时间中的分钟分量
22	MONTH(date)	返回日期中的月份分量
23	MONTHNAME(date)	返回日期中月分量的名称
24	MONTHS_BETWEEN(date1,date2)	返回两个日期之间的月份数
25	NEXT_DAY(date1,char2)	返回输入日期指定若干天后的日期
26	NOW()	返回系统当前时间戳
27	QUARTER(date)	返回日期在所处年中的季节数
28	SECOND(time)	返回时间中的秒分量
29	ROUND (date1,char2)	把日期四舍五入到最接近格式元素指定的形式
30	TIMESTAMPADD(interval,n,timestamp)	返回时间 timestamp 加上 n 个 interval 类型时间间隔的结果
31	TIMESTAMPDIFF(interval,timeStamp1,timestamp2)	返回一个表明 timestamp2 与 timestamp1 之间的 interval 类型时间间隔的整数
32	SYSDATE()	返回系统的当前日期
33	TO_DATE(CHAR[,fmt])	字符串转换为日期数据类型
34	TRUNC(date[,format])	把日期截断到最接近格式元素指定的形式
35	WEEK(date)	返回日期为所在年中的第几周
36	WEEKDAY(date)	返回当前日期的星期值
37	WEEKS_BETWEEN(date1,date2)	返回两个日期之间相差周数
38	YEAR(date)	返回日期的年分量
39	YEARS_BETWEEN(date1,date2)	返回两个日期之间相差年数

空值判断函数

序号	函数名	功能简要说明
01	COALESCE(n1,n2,…nx)	返回第一个非空的值
02	IFNULL(n1,n2)	返回第一个非空的值
03	ISNULL(n1,n2)	使用指定的替换值替换 NULL

04	NULLIF(n1,n2)	如果 n1=n2 返回 NULL，否则返回 n1
05	NVL(n1,n2)	返回第一个非空的值

类型转换函数

序号	函数名	功能简要说明
01	CAST(value AS 类型说明)	将 value 转换为指定的类型
02	CONVERT(类型说明,value)	将 value 转换为指定的类型

杂类函数

序号	函数名	功能简要说明
01	DECODE(exp, search1, result1, … searchn, resultn, [default])	查表译码

系统函数

序号	函数名	功能简要说明
01	CUR_DATABASE()	返回数据库名
02	DBID()	返回当前 Database ID
03	EXTENT()	取得当前每个簇中的块数
04	PAGE()	取得当前每个块的字节数
05	SESSID()	取得当前 Session ID
06	UID()	返回当前用户 ID
07	USER()	返回当前连接的用户名
08	TABLEDEF(db,schema,table)	返回基表的建表语句
09	VSIZE()	求某个对象的内部存储大小
10	SET_TABLE_OPTION ([[db.] schema.]tablename, option, value)	设置基表选项值
11	SET_INDEX_OPTION([[db.].schema.]indexname, option, value)	设置索引选项值
12	UNLOCK_USER (LoginName)	解除对登录的锁定
13	GET_AUDIT()	查询当前系统的审计开关状态
14	SET_AUDIT(n)	设置系统的审计开关状态

8.1　数值函数

数值函数接受数值参数并返回数值作为结果，BIT 类型能够等同于数值类型出现在需要数值类型参数的函数中。

1．函数 ABS

语法：ABS(n)

功能：返回 n 的绝对值。n 必须是数值类型。

例：假定当前数据库为 CW，用户 USER1 查询价格小于 1000 元或大于 2000 元的商品信息。

```
SELECT　商品编号,商品名,价格,仓库地址,厂商编号　FROM　商品登记
WHERE　ABS(价格-1500)>500;
```

查询结果如下表 8.1.1 所示。

表 8. 1. 1 查询结果

商品编号	商品名	价格	仓库地址	厂商编号
C0001	电视机	3000.00	武汉	B0A01
C0003	鞋_1	50.00	武汉	B0003
C0004	电视机	2500.00	上海	B0A04
C0006	鞋_2	500.00	武汉	B0003
C0007	电视机	2800.00	北京	B0A04

2．函数 ACOS

语法：ACOS(n)

功能：返回 n 的反余弦值。n 必须是数值类型，且取值在-1 到 1 之间，函数结果从 0 到π。

例：

```
SELECT acos(0);
```

查询结果为：1.570796

3．函数 ASIN

语法：ASIN(n)

功能：返回 n 的反正弦值。 n 必须是数值类型， 且取值在-1 到 1 之间， 函数结果从-π/2 到π/2。

例：

```
SELECT asin(0);
```

查询结果为：0.000000

4．函数 ATAN

语法：ATAN(n)

功能：返回 n 的反正切值。n 必须是数值类型，取值可以是任意大小，函数结果从-π/2 到π/2。

例：

```
SELECT atan(1);
```

查询结果为：0.785398

5．函数 ATAN2

语法：ATAN2(n, m)

功能：返回 n/m 的反正切值。n,m 必须是数值类型，取值可以是任意大小，函数结果从-π/2 到π/2。

例：

```
SELECT atan2(0.2,0.3);
```

查询结果为：0.588003

6．函数 CEIL

语法：CEIL(n)

功能：返回大于等于 n 的最小整数。n 必须是数值类型。返回类型与 n 的类型相同。

例：

```
SELECT    CEIL(15.6);
```

查询结果为：16.0

```
SELECT    CEIL(-16.23);
```

查询结果为：-16.0

7．函数 CEILING

语法：CEILING(n)

功能：返回大于等于 n 的最小整数。等价于函数 CEIL()。

8．函数 COS

语法：COS(n)

功能：返回 n 的余弦值。n 必须是数值类型，是用弧度表示的值。将角度乘以π/180，可以转换为弧度值。

例：

```
SELECT cos(14.78);
```

查询结果为：-0.599465

9．函数 COT

语法：COT(n)

功能：返回 n 的余切值。n 必须是数值类型，是用弧度表示的值。将角度乘以π/180，可以转换为弧度值。

例：

```
SELECT COT(20 * 3.1415926/180);
```

查询结果为：2.7474846892187106

10．函数 COSH

语法：COSH(n)

功能：返回 n 的双曲余弦值。

例：

```
SELECT COSH(0)"Hyperbolic cosine of 0";
```

查询结果为：1.000000

11．函数 DEGREES

语法：DEGREES(n)

功能：返回弧度 n 对应的角度值，返回值类型与 n 的类型相同。

例：

```
SELECT DEGREES(1.0);
```

查询结果为：57.295780

12．函数 EXP

语法：EXP(n)

功能：返回 e 的 n 次幂。

例：

```
SELECT EXP(4) "e to the 4th power";
```

查询结果为：54.598150

13．函数 FLOOR

语法：FLOOR(n)

功能：返回小于等于 n 的最大整数值。n 必须是数值类型。返回类型与 n 的类型相同。

例：

```
SELECT   FLOOR(15.6);
```

查询结果为：15.0

```
SELECT   FLOOR(-16.23);
```

查询结果为：-17.0

14．函数 LN

语法：LN(n)

功能：返回 n 的自然对数。n 为数值类型，且大于 0。

例：

```
SELECT ln(95) "Natural log of 95";
```

查询结果为：4.553877

15．函数 LOG

语法：LOG(m[,n])

功能：返回数值 m 以数值 n 为底的对数；若参数 n 省略，返回 m 的自然对数。m,n 为数值类型，n 大于 0 且不为 1。

例：

```
SELECT LOG(100,10);
```

查询结果为：2.0

```
SELECT LOG(95);
```

查询结果为：4.553877

16．函数 LOG10

语法：LOG10(n)

功能：返回数值 n 以 10 为底的对数。n 为数值类型，且大于 0。

例：

```
SELECT LOG10(100);
```

查询结果为：2.000000

17．函数 MOD

语法：MOD(m,n)

功能：返回以 m 除以 n 的余数。m,n 为数值类型，n 不能为 0。

例：

```
SELECT  商场编号,商品编号, 采购商品数量-120,
MOD(采购商品数量-120,6),进货日期　FROM　供货登记;
```

查询结果如下表 8.1.2 所示。

表 8. 1. 2 查询结果

厂商编号	商品编号	采购商品数量-120	MOD(采购商品数量-120，6)	进货日期
A0001	C0002	-20	-2	1998-10-05
A0001	C0004	-70	-4	1998-10-18
A0002	C0003	880	4	1998-10-20
A0002	C0002	-90	0	1998-10-05
A0003	C0002	-110	-2	1998-10-31
A0004	C0005	80	2	1998-10-20
A0002	C0002	-20	-2	1998-10-30

18．函数 PI

语法：PI()

功能：返回常数 π 。

例：

```
SELECT PI();
```

查询结果为：3.141593

19．函数 POWER

语法：POWER(m,n)

功能：返回 m 的 n 次幂。m,n 为数值类型，如果 m 为负数的话，n 必须为一个整数。

例：

```
SELECT POWER(3,2) "Raised";
```

查询结果为：9.000000

```
SELECT POWER(-3,3) "Raised";
```

查询结果为：-27.000000

20．函数 RADIANS()

语法：RADIANS(n)

功能：返回角度 n 对应的弧度值，返回值类型与 n 的类型相同。

例：

```
SELECT RADIANS(180.0);
```

查询结果为：3.141593

21．函数 RAND()

语法：RAND([n])

功能：返回一个 0 到 1 之间的随机浮点数。n 为数值类型，为生成随机数的种子，当 n 省略时，系统自动生成随机数种子。

例：

```
SELECT RAND();
```

查询结果为一个随机生成的小数

```
SELECT RAND(314);
```

查询结果为：0.032472

22．函数 ROUND

语法：ROUND(n [,m])

功能：返回四舍五入到小数点后面 m 位的 n 值。m 应为一个整数，缺省值为 0，m 为负整数则四舍五入到小数点的左边，m 为正整数则四舍五入到小数点的右边。若 m 为小数，系统将自动将其转换为整数。

例：

```
SELECT  厂商编号,厂商名,资产总值/6, ROUND(资产总值/6,1),厂商地址 FROM 厂商登记;
```

查询结果如下表 8.1.3 所示。

表 8.1.3 查询结果

厂商编号	厂商名	资产总值/6	ROUND(资产总值/6,1)	厂商地址
B0A01	华乐电视机厂	8.33	8.3	北京
B0A02	日东冰箱厂	13.33	13.3	北京
B0003	万里鞋厂	5.00	5.0	北京
B0A04	海天电视机厂	10.00	10.0	武汉
B0A05	小鸭洗衣机厂	166.67	166.7	上海
B0006	美的服装厂	3.33	3.3	武汉

```
SELECT    ROUND(15.163,-1);
```

查询结果为：20

```
SELECT    ROUND(15.163);
```

查询结果为：15

23．函数 SIGN

语法：SIGN(n)

功能：如果 n 为正数，SIGN(n)返回 1，如果 n 为负数，SIGN(n)返回-1，如果 n 为 0，SIGN(n)返回 0。

例：

```
SELECT  商场编号,商品编号,  采购商品数量-50,
SIGN(采购商品数量-50),进货日期  FROM  供货登记;
```

查询结果如下表 8.1.4 所示。

表 8.1.4 查询结果

商场编号	商品编号	采购商品数量-50	SIGN(采购商品数量-50)	进货日期
A0001	C0002	50	1	1998-10-05
A0001	C0004	0	0	1998-10-18
A0002	C0003	950	1	1998-10-20
A0002	C0002	-20	-1	1998-10-05
A0003	C0002	-40	-1	1998-10-31

A0004	C0005	150	1	1998-10-20
A0002	C0002	50	1	1998-10-30

24．函数 SIN

语法：SIN(n)

功能：返回 n 的正弦值。n 必须是数值类型，是用弧度表示的值。将角度乘以π/180，可以转换为弧度值。

例：

```
SELECT SIN(0);
```

查询结果为：0.000000

25．函数 SINH

语法：SINH(n)

功能：返回 n 的双曲正弦值。

例：

```
SELECT SINH(1);
```

查询结果为：1.175201

26．函数 SQRT

语法：SQRT(n)

功能：返回 n 的平方根。n 为数值类型，且大于等于 0。

例：

```
SELECT 厂商编号,厂商名,资产总值,SQRT(资产总值),厂商地址 FROM 厂商登记;
```

查询结果如下表 8.1.5 所示。

表 8.1.5 查询结果

厂商编号	厂商名	资产总值	SQRT(资产总值)	厂商地址
B0A01	华乐电视机厂	50.00	7.071068	北京
B0A02	日东冰箱厂	80.00	8.944272	武汉
B0003	万里鞋厂	30.00	5.477226	北京
B0A04	海天电视机厂	60.00	7.745967	武汉
B0A05	小鸭洗衣机厂	1000.00	31.622777	上海
B0006	美的服装厂	20.00	4.472136	武汉

27．函数 TAN

语法：TAN(n)

功能：返回 n 的正切值。n 必须是数值类型，是用弧度表示的值。将角度乘以π/180，可以转换为弧度值。

例：

```
SELECT TAN(45*Pi()/180);
```

查询结果为：1.000000

28．函数 TANH

语法：TANH(n)

功能：返回 n 的双曲正切值。

例：

```
SELECT TANH(0);
```

查询结果为：0.000000

29．函数 TRUNC

语法：TRUNC(n [,m])

功能：将数值 n 的小数点后第 m 位以后的数全部截去。当数值参数 m 省略时，m 默认为 0。当数值参数 m 为负数时表示将数值 n 的小数点前的第 m 位截去。

例：

```
SELECT  厂商编号,厂商名,SQRT(资产总值),TRUNC(SQRT(资产总值),2), 厂商地址
FROM  厂商登记;
```

查询结果如下表 8.1.6 所示。

表 8. 1. 6 查询结果

厂商编号	厂商名	SQRT(资产总值)	TRUNC(SQRT(资产总值),2)	厂商地址
B0A01	华乐电视机厂	7.071068	7.07	北京
B0A02	日东冰箱厂	8.944272	8.94	北京
B0003	万里鞋厂	5.477226	5.47	北京
B0A04	海天电视机厂	7.745967	7.74	武汉
B0A05	小鸭洗衣机厂	31.622777	31.62	上海
B0006	美的服装厂	4.472136	4.47	武汉

```
SELECT   TRUNC(15.167,-1);
```

查询结果为：10

30．函数 TRUNCATE

语法：TRUNCATE(n [,m])

功能：将数值 n 的小数点后第 m 位以后的数全部截去。当数值参数 m 省略时，m 默认为 0。当数值参数 m 为负数时表示将数值 n 的小数点前的第 m 位截去。等价于 TRUNC()。

8.2　字符串函数

字符串函数一般接受字符类型（包括 CHAR 和 VARCHAR）和数值类型的参数，返回值一般是字符类型或是数值类型。

1．函数 ASCII

语法：ASCII(char)

功能：返回字符 char 的 ASCII 码。

例：

```
SELECT   ASCII('B');
```

查询结果为：66

2．函数 BIT_LENGTH

语法：BIT_LENGTH(char)

功能：返回字符串的位（bit）长度。

例：

```
SELECT BIT_LENGTH('ab');
```

查询结果为：16

3．函数 CHAR

语法：CHAR(n)

功能：返回 ASCII 码为 n 的字符。n 范围为 0－255。

例：

```
SELECT    CHAR(66),CHAR(67),CHAR(68);
```

查询结果为：B C D

4．函数 CHAR_LENGTH / CHARACTER_LENGTH

语法：CHAR_LENGTH(char) 或 CHARACTER_LENGTH(char)

功能：返回字符串 char 的长度，以字符作为计算单位，一个汉字作为一个字符计算。字符串尾部的空格也计数。

例：

```
SELECT   厂商名,CHAR_LENGTH(TRIM(BOTH ' ' FROM 厂商名))
FROM 厂商登记;
```

查询结果如下表 8.2.1 所示。

表 8.2.1 查询结果

厂商名	CHAR_LENGTH(TRIM(BOTH ' ' FROM 厂商名))
华乐电视机厂	6
日东冰箱厂	5
万里鞋厂	4
海天电视机厂	6
小鸭洗衣机厂	6
美的服装厂	5

```
SELECT CHAR_LENGTH('我们');
```

查询结果为：2

5．函数 CHR

语法：CHR(n)

功能：返回 ASCII 码为 n 的字符。等价于 CHAR()。

6．函数 CONCAT

语法：CONCAT(char1,char2)

功能：返回字符串 char1 串接字符串 char2 的结果，该函数等价于运算符||。

例：

```
SELECT 厂商编号,厂商名,CONCAT(厂商编号,厂商名) FROM 厂商登记;
```

查询结果如下表 8.2.2 所示。

表 8.2.2 查询结果

厂商编号	厂商名	CONCAT(厂商编号,厂商名)
B0A01	华乐电视机厂	B0A01 华乐电视机厂
B0A02	日东冰箱厂	B0A02 日东冰箱厂
B0003	万里鞋厂	B0003 万里鞋厂
B0A04	海天电视机厂	B0A04 海天电视机厂
B0A05	小鸭洗衣机厂	B0A05 小鸭洗衣机厂
B0006	美的服装厂	B0006 美的服装厂

7．函数 DIFFERENCE()

语法：DIFFERENCE(char1,char2)

功能：以整数返回两个字符串的 SOUNDEX 值之差。

例：

```
SELECT DIFFERENCE('she', 'he');
```

查询结果为：2

8．函数 INITCAP

语法：INITCAP(char)

功能：返回句子字符串中，每一个单词的第一个字母改为大写，其他字母改为小写。单词用空格分隔，不是字母的字符不受影响。

例：

```
SELECT INITCAP('hello world');
```

查询结果为：Hello World

9．函数 INSERT / INSSTR

语法： INSERT(char1,n1,n2,char2) / INSSTR(char1,n1,n2,char2)

功能：将字符串 char1 从 n1 的位置开始删除 n2 个字符，并将 char2 插入到 char1 中 n1 的位置。

例：

```
SELECT INSERT('That is a cake',2,3, 'his') ;
```

查询结果为：This is a cake

10．函数 INSTR

语法：INSTR(char1,char2[n,[m]])

功能：返回 char1 中包含 char2 的特定位置。INSTR 从 char1 的左边开始搜索，开始位置是 n，如果 n 为负数，则搜索从 char1 的最右边开始，当搜索到 char2 的第 m 次出现时，返回所在位置。n 和 m 的缺省值都为 1，即返回 char1 中第一次出现 char2 的位置，这时与 POSITION 相类似。如果从 n 开始没有找到第 m 次出现的 char2，则返回 0。n 和 m 以字符作为计算单位，一个西文字符和一个汉字都作为一个字符计算。

此函数中 char1 可以是 CHAR、VARCHAR 和 TEXT 数据类型，char2 是 CHAR 或 VARCHAR 类型。n 和 m 是数值类型。

例：

```
SELECT INSTR('CORPORATE FLOOR', 'OR', 3, 2) "Instring";
```

查询结果为：14

```
SELECT INSTR('我们的计算机', '计算机',1,1);
```

查询结果为：4

11．函数 INSTRB

语法：INSTRB(char1,char2[n,[m]])

功能：返回 char1 中包含 char2 的特定位置。INSTR 从 char1 的左边开始搜索，开始位置是 n，如果 n 为负数，则搜索从 char1 的最右边开始，当搜索到 char2 的第 m 次出现时，返回所在位置。n 和 m 的缺省值都为 1，即返回 char1 中第一次出现 char2 的位置，这时与 POSITION 相类似。如果从 n 开始没有找到第 m 次出现的 char2，则返回 0。以字节作为计算单位，一个汉字作为两个字节计算。

此函数 char1 可以是 CHAR,VARCHAR 和 TEXT 数据类型，char2 是 CHAR 或 VARCHAR 类型。n 和 m 是数值类型。

例：

```
SELECT INSTRB('CORPORATE FLOOR', 'OR', 3, 2) "Instring";
```

查询结果为：14

```
SELECT INSTRB('我们的计算机', '计算机',1,1);
```

查询结果为：7

12．函数 LCASE

语法：LCASE(char)

功能：返回字符串中，所有字母改为小写，不是字母的字符不受影响。

例：

```
SELECT   厂商编号,LCASE(厂商编号)   FROM   厂商登记;
```

查询结果如下表 8.2.3 所示。

表 8.2.3 查询结果

厂商编号	LCASE (厂商编号)
B0A01	b0a01
B0A02	b0a02
B0003	b0003
B0A04	b0a04
B0A05	b0a05
B0006	b0006

13．函数 LEFT / LEFTSTR

语法：LEFT(char,n) / LEFTSTR(char,n)

功能：返回字符串最左边的 n 个字符组成的字符串。

例：

```
SELECT   厂商编号,厂商名,LEFT (厂商名,2)   FROM   厂商登记;
```

查询结果如下表 8.2.4 所示。

表 8.2.4 查询结果

厂商编号	厂商名	LEFT(厂商名,2)
B0A01	华乐电视机厂	华乐
B0A02	日东冰箱厂	日东
B0003	万里鞋厂	万里
B0A04	海天电视机厂	海天
B0A05	小鸭洗衣机厂	小鸭
B0006	美的服装厂	美的

```
SELECT　LEFT ('computer science',10);
```

查询结果为：computer s

14．函数 LENGTH

```
语法：LENGTH(char)
```

功能：返回给定字符串表达式的字符（而不是字节）个数，其中不包含尾随空格。

例：

```
SELECT LENGTH('hi,你好□□');
```

查询结果为：7

说明：□表示空格字符

15．函数 LENGTHB / OCTET_LENGTH

```
语法：LENGTHB(char)/ OCTET_LENGTH(char)
```

功能：返回字符串 char 的长度，以字节作为计算单位，一个汉字作为两个字节计算。

例：

```
SELECT LENGTHB('ＣＡＮＤＩＤＥ') "Length in bytes";
```

查询结果为：14

16．函数 LOCATE

```
语法：LOCATE(char1,char2[,n])
```

功能：返回字符串 char1 在 char2 中从位置 n 开始首次出现的位置，如果参数 n 省略或为负数，则从 char2 的最左边开始找。

例：

```
SELECT LOCATE('man', 'The manager is a man', 10);
```

查询结果为：18

```
SELECT LOCATE('man', 'The manager is a man');
```

查询结果为：5

17．函数 LOWER

```
语法：LOWER(char)
```

功能：返回字符串中，所有字母改为小写，不是字母的字符不受影响。等价于 LCASE()。

18．函数 LPAD

```
语法：LPAD(char1,length[,char2])
```

功能：返回值为字符串 char1 左边增加 char2，长度达到 length 的字符串。如果未指定 char2，缺省值为空格。如果 char2 的长度比 length 小，则根据需要进行复制；反之，则仅仅复制 char2 的前 length 个字符。长度以字节作为计算单位，一个汉字作为两个字节计算。

例：

```
SELECT LPAD(LPAD('FX',19,'Teacher'),22,'BIG') "LPAD example";
```

查询结果为：BIGTeacherTeacherTeaFX

```
SELECT LPAD('计算机',15, '我们的');
```

查询结果为：我们的我眉扑慊□

```
SELECT LPAD('计算机',16, '我们的');
```

查询结果为：我们的我们计算机

19．函数 LTRIM

语法：LTRIM(char1[,set])

功能：删除字符串 char 左边起出现的 set 中的任何字符，当遇到不在 set 中的第一个字符时结果被返回。set 缺省为空格。

例：

```
SELECT LTRIM('xyyxxxXxyLAST WORD', 'xy') "LTRIM example";
```

查询结果为：XxyLAST WORD

```
SELECT LTRIM('我们的计算机', '我们');
```

查询结果为：的计算机

20．函数 POSITION

语法：POSITION(char1 IN char2) / POSITION(char1, char2)

功能：返回在 char2 串中第一次出现的 char1 的位置，如果 char1 是一个零长度的字符串，POSITION 返回 1，如果 char2 中 char1 没有出现，则返回 0。以字节作为计算单位，一个汉字作为两个字节计算。

例：

```
SELECT   厂商名,POSITION('厂' IN 厂商名)   FROM   厂商登记;
```

查询结果如下表 8.2.5 所示。

表 8.2.5 查询结果

厂商名	POSITION('厂' IN 厂商名)
华乐电视机厂	11
日东冰箱厂	9
万里鞋厂	7
海天电视机厂	11
小鸭洗衣机厂	11
美的服装厂	9

21．函数 REPEAT / REPEATSTR

语法：REPEAT(char,n) / REPEATSTR(char,n)

功能：返回将字符串重复 n 次形成的字符串。

例：

```
SELECT REPEAT ('Hello ',3);
```

查询结果为：Hello Hello Hello

22．函数 REPLACE

语法：REPLACE(char, search [,replacement])

功能：REPLACE 的三个参数都是字符串类型，返回在 char 中用 replacement 置换 search 的结果，如果 replacement 没有被指定，则在 char 中出现的 search 都将被删除。

例：

```
SELECT  厂商编号,厂商名,REPLACE(厂商名, '厂', '集团公司') FROM  厂商登记;
```

查询结果如下表 8.2.6 所示。

表 8.2.6 查询结果

厂商编号	厂商名	REPLACE(厂商名, '厂', '集团公司')
B0A01	华乐电视机厂	华乐电视机集团公司
B0A02	日东冰箱厂	日东冰箱集团公司
B0003	万里鞋厂	万里鞋集团公司
B0A04	海天电视机厂	海天电视机集团公司
B0A05	小鸭洗衣机厂	小鸭洗衣机集团公司
B0006	美的服装厂	美的服装集团公司

23.函数 REVERSE

语法：reverse(char)

功能：将输入字符串 string 的字符顺序反转后返回。

例：

```
SELECT REVERSE('abcd');
```

查询结果: dcba

24．函数 RIGHT / RIGHTSTR

语法：RIGHT(char,n) / RIGHTSTR(char,n)

功能：返回字符串最右边 n 个字符组成的字符串。

例：

```
SELECT  厂商编号,RIGHT (厂商编号,4) FROM  厂商登记;
```

查询结果如下表 8.2.7 所示。

表 8.2.7 查询结果

厂商编号	RIGHT(厂商编号,4)
B0A01	0A01
B0A02	0A02
B0003	0003
B0A04	0A04
B0A05	0A05
B0006	0006

```
SELECT RIGHTSTR('computer',3);
```

查询结果为：ter

25．函数 RPAD

语法：RPAD(char1,length[,char2])

功能：返回值为字符串 char1 右边增加 char2，长度达到 length 的字符串。如果未指定 char2，缺省值为空格。如果 char2 的长度比 length 小，则根据需要进行复制；反之，则仅仅复制 char2 的前 length 个字符。以字节作为计算单位，一个汉字作为两个字节计算。

例：

```
SELECT RPAD('FUXIN',11, 'BigBig') "RPAD example";
```

查询结果为：FUXINBigBig

26．函数 RTRIM

语法：RTRIM(char1[,set])

功能：删除字符串 char 右边起出现的 set 中的任何字符，当遇到不在 set 中的第一个字符时结果被返回。set 缺省为空格。

例：

```
SELECT RTRIM('TURNERyXxxxyyyxy', 'xy') "RTRIM e.g.";
```

查询结果为：TURNERyX

```
SELECT RTRIM('我们的计算机','我计算机');
```

查询结果为：我们的

27．函数 SOUNDEX

语法：SOUNDEX(char)

功能：返回一个表示英文字符串发音的字符串，仅仅对英文字符串有效，对别的字符或数字返回 0000。

例：

```
SELECT SOUNDEX('Hello');
```

查询结果为：H120

28．函数 SPACE

语法：SPACE(n)

功能：返回一个包含 n 个空格的字符串。

例：

```
SELECT SPACE(5);
```

查询结果为：□□□□□

```
SELECT CONCAT(CONCAT('Hello',SPACE(3)), 'world');
```

查询结果为：Hello□□□world

说明：□表示空格字符

29．函数 SUBSTR

语法：SUBSTR(char,m[,n]) / SUBSTRING(char from m [for n])

功能：返回 char 中从字符位置 m 开始的 n 个字符。若 m 为 0，则把 m 就当作 1 对

待。若 m 为正数，则返回的字符串是从左边到右边计算的；反之，返回的字符是从 char 的结尾向左边进行计算的。如果没有给出 n，则缺省的设置为整个字符串. 如果 n 小于 1，则返回 NULL。以字符作为计算单位，一个西文字符和一个汉字都作为一个字符计算。

例：

SELECT　厂商编号,厂商名,SUBSTRING(厂商名 FROM 3 FOR 6) FROM　厂商登记;

查询结果如下表 8.2.8 所示。

表 8.2.8 查询结果

厂商编号	厂商名	SUBSTRING(厂商名 FROM 3 FOR 6)
B0A01	华乐电视机厂	电视机厂
B0A02	日东冰箱厂	冰箱厂
B0003	万里鞋厂	鞋厂
B0A04	海天电视机厂	电视机厂
B0A05	小鸭洗衣机厂	洗衣机厂
B0006	美的服装厂	服装厂

SELECT SUBSTR('我们的计算机',3,4) "Subs";

查询结果为：的计算机

30．函数 SUBSTRB

语法：SUBSTRB(char,m[,n])

功能：返回 char 中从字节位置 m 开始的 n 个字节。若 m 为 0，则 m 就当作 1 对待。若 m 为正数，则返回的字符串是从左边到右边计算的；反之，返回的字符是从 char 的结尾向左边进行计算的。如果没有给出 n，则缺省的设置为整个字符串. 如果 n 小于 1，则返回 NULL。以字节作为计算单位，一个汉字作为两个字节计算。

例：

SELECT SUBSTRB('ABCDEFG',3,3) "Subs";

查询结果为：CDE

SELECT SUBSTRB('我们的计算机',3,4) "Subs";

查询结果为：们的

31．函数 TO_CHAR

语法：TO_CHAR(DATE[,fmt])

功能：将日期数据类型 DATE 转换为一个在日期语法 fmt 中指定语法的 VARCHAR 类型字符串。若没有指定语法，日期 DATE 将按照缺省的语法转换为一个 VARCHAR 值。

有效的日期语法分量如下表：

有效日期语法分量

元素	说明
- / , . :	标点符号在结果中重新复制
D	月中的某一天（1－31）
DD	

DDD	
HH HH12 HH24	天中的时（0－23） （均为 24 小时制）
MI	分（0－59）
MM	月（01－12）
SS	秒（0－59）
YYYY	4 位的年份
YY	年份的最后 2 位数字

DM 缺省的日期语法为：'YYYY-MM-DD HH:MI:SS'。

例：

```
SELECT TO_CHAR(SYSDATE,'YYYYMMDD');
```

查询结果：20030619　/* 假设本查询操作发生在 2003-06-19 09:00*/

```
SELECT TO_CHAR(SYSDATE,'YYYY/MM/DD');
```

查询结果：2003/06/19

```
SELECT TO_CHAR(SYSDATE,'HH24:MI');
```

查询结果：09:02

```
SELECT TO_CHAR(SYSDATE,'YYYYMMDD HH24:MI:SS');
```

查询结果：20030619 09:03:16

```
SELECT TO_CHAR(SYSDATE,'YYYY-MM-DD HH24:MI:SS');
```

查询结果：2003-06-19 09:03:55

32．函数 TRANSLATE

```
语法：TRANSLATE(char,from,to)
```

功能：此函数查看 char 中的每一个字符，然后检查 from 以确定该字符是否在其中。如果在的话，TRANSLATE 记下该字符在 from 中的位置，然后查找 to 中相同的位置，无论找到什么字符，用其替换 char 中的字符。

例：

```
SELECT　厂商编号,TRANSLATE (厂商编号,
'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'9876543210abcdefghijklmnopqrstuvwxyz') FROM  厂商登记;
```

查询结果如下表 8.2.9 所示。

表 8.2.9 查询结果

厂商编号	TRANSLATE(厂商编号,…,…)
B0A01	b9a98
B0A02	b9a97
B0003	b9996
B0A04	b9a95
B0A05	b9a94
B0006	b9993

```
SELECT TRANSLATE('我们的计算机', '我们的', '你们的');
```

查询结果为：你们的计算机。

33．函数 TRIM

语法：TRIM([[LEADING | TRAILING | BOTH] [char1] FROM] char2)

功能：TRIM 从 char2 的首端(LEADING)或末端(TRAILING)或两端(BOTH)删除 char1 字符，如果任何一个变量是 NULL，则返回 NULL。默认的修剪方向为 BOTH，默认的修剪字符为空格。

例：

```
SELECT  厂商编号,TRIM(LEADING 'B' FROM 厂商编号),
TRIM(TRAILING '1' FROM 厂商编号)  FROM  厂商登记;
```

查询结果如下表 8.2.10 所示。

表 8. 2. 10 查询结果

厂商编号	TRIM(LEADING 'B' FROM 厂商编号)	TRIM(TRAILING '1' FROM 厂商编号)
B0A01	0A01	B0A0
B0A02	0A02	B0A02
B0003	0003	B0003
B0A04	0A04	B0A04
B0A05	0A05	B0A05
B0006	0006	B0006

```
SELECT  TRIM( '   Hello World   ');
```

查询结果为：Hello World

```
SELECT  TRIM(LEADING FROM '   Hello World   ');
```

查询结果为：Hello World□□□

说明：□表示空格字符，下同。

```
SELECT  TRIM(TRAILING FROM '   Hello World   ');
```

查询结果为：□□□Hello World

```
SELECT  TRIM(BOTH FROM '   Hello World   ');
```

查询结果为：Hello World

34．函数 UCASE

语法：UCASE(char)

功能：返回字符串中，所有字母改为大写，不是字母的字符不受影响。

例：

```
SELECT  UCASE('hello world');
```

查询结果为：HELLO WORLD

35．函数 UPPER

语法：UPPER(char)

功能：返回字符串中，所有字母改为大写，不是字母的字符不受影响。等价于 UCASE()。

8.3　日期时间函数

日期时间函数的参数至少有一个是日期时间类型（TIME，DATE，TIMESTAMP），返回值一般为日期时间类型和数值类型。

1．函数 ADD_DAYS

语法：ADD_DAYS(date, n)

功能：返回日期 date 加上相应天数 n 后的日期值。n 可以是任意整数，date 是日期类型(DATE)或时间戳类型(TIMESTAMP)，返回类型与 date 相同。

例：

```
SELECT ADD_DAYS( DATE '2000-01-12',1);
```

查询结果: 2000-01-13;

2．函数 ADD_MONTHS

语法：ADD_MONTHS(date,n)

功能：返回日期 date 加上 n 个月的日期时间值。n 可以是任意整数，date 是日期类型(DATE)或时间戳类型(TIMESTAMP)，返回类型与 date 相同。如果相加之后的结果日期中月份所包含的天数比 date 日期中的日分量要少，那么结果日期的该月最后一天被返回。如果 date 是 TIMESTAMP 类型，结果值的时间分量不变。

例：

```
SELECT ADD_MONTHS(DATE '2000-01-31',1);
```

查询结果为：2000-02-29

```
SELECT ADD_MONTHS(TIMESTAMP '2000-01-31 20:00:00',1);
```

查询结果为：2000-02-29 20:00:00

3．函数　ADD_WEEKS

语法：ADD_WEEKS(date, n)

功能：返回日期 date 加上相应星期数 n 后的日期值。n 可以是任意整数，date 是日期类型(DATE)或时间戳类型(TIMESTAMP)，返回类型与 date 相同。

例：

```
SELECT ADD_WEEKS( DATE '2000-01-12',1);
```

查询结果: 2000-01-19;

4．函数 CURDATE

语法：CURDATE()

功能：返回当前日期值，结果类型为 DATE。

例：

```
SELECT     CURDATE();
```

查询结果为：执行此查询当天日期，如 2003-02-27

5．函数 CURTIME

语法：CURTIME([n])

功能：返回当前时间值，结果类型为 TIME。n 表示秒精度，即返回值中秒分量小数点后面的位数。

例：

```
SELECT   CURTIME();
```

查询结果为：执行此查询的当前时间，如 11:38:34

```
SELECT   CURTIME(3);
```

查询结果为：执行此查询的当前时间，3 表示秒精度，如 11:38:34.000

6．函数 CURRENT_DATE

语法：CURRENT_DATE

功能：返回当前日期值，结果类型为 DATE。等价于 CURDATE()。

7．函数 CURRENT_TIME

语法：CURRENT_TIME[(n)]

功能：返回当前时间值，结果类型为 TIME。n 表示秒精度，即返回值中秒分量小数点后面的位数。等价于 CURTIME()。

8．函数 CURRENT_TIMESTAMP

语法：CURRENT_TIMESTAMP(n)

功能：返回当前日期时间值，结果类型为 TIMESTAMP。n 表示秒精度，即返回值中秒分量小数点后面的位数。

例：

```
SELECT    CURRENT_TIMESTAMP(2);
```

查询结果为：执行此查询的当前日期时间，如 2003-02-27 13:03:56.00

9．函数 DATEADD

语法：DATEADD(datepart,n,date)

功能：向指定的日期 date 加上 n 个 datepart 指定的时间段，返回新的 timestamp 值。datepart 可以为 YEAR（缩写 YY 或 YYYY）、QUARTER（缩写 QQ 或 Q）、MONTH（缩写 MM 或 M）、DAYOFYEAR（缩写 DY 或 Y）、DAY（缩写 DD 或 D）、WEEK（缩写 WK 或 WW）、HOUR（缩写 HH）、MINUTE（缩写 MI 或 N）、SECOND（缩写 SS 或 S）和 MILLISECOND（缩写 MS）。

10．函数 DATEDIFF

语法：DATEDIFF(datepart,date1,date2)

功能：返回跨两个指定日期的日期和时间边界数。datepart 可以为 YEAR（缩写 YY 或 YYYY）、QUARTER（缩写 QQ 或 Q）、MONTH（缩写 MM 或 M）、DAYOFYEAR（缩写 DY 或 Y）、DAY（缩写 DD 或 D）、WEEK（缩写 WK 或 WW）、HOUR（缩写 HH）、MINUTE（缩写 MI 或 N）、SECOND（缩写 SS 或 S）和 MILLISECOND（缩写 MS）。

注：当结果超出整数值范围，DATEDIFF 产生错误。对于毫秒 MILLISECOND，最大数可以是小于 2^{64} 的时间间隔，这一间隔已经超过了 DATE 数据类型所能接受的最大范围，故而可以认为是任意长时间。对于秒 SECOND，最大数是 68 年。

例：

```
SELECT DATEDIFF(QQ, '2003-06-01', DATE '2002-01-01');
```

查询结果为：-5

```
SELECT DATEDIFF(MONTH, '2001-06-01', DATE '2002-01-01');
```

查询结果为：7

```
SELECT DATEDIFF(WK, DATE '2003-02-07',DATE '2003-02-14');
```

查询结果为：1

```
SELECT DATEDIFF(MS,'2003-02-14 12:10:10.000','2003-02-14 12:09:09.300');
```

查询结果为：-60700

11．函数 DATEPART

语法：DATEPART(datepart,date)

功能：返回代表日期的指定部分的整数。datepart 可以为 YEAR（缩写 YY 或 YYYY）、QUARTER（缩写 QQ 或 Q）、MONTH（缩写 MM 或 M）、DAYOFYEAR（缩写 DY 或 Y）、DAY（缩写 DD 或 D）、WEEK（缩写 WK 或 WW）、WEEKDAY（缩写 DW）、HOUR（缩写 HH）、MINUTE（缩写 MI 或 N）、SECOND（缩写 SS 或 S）和 MILLISECOND（缩写 MS）。

例：

```
SELECT DATEPART(SECOND, DATETIME '2000-02-02 13:33:40.00');
```

查询结果为：40

```
SELECT DATEPART(DY, '2000-02-02');
```

查询结果为：33

```
SELECT DATEPART(WEEKDAY, '2002-02-02');
```

查询结果为：7

12．函数 DAYNAME

语法：DAYNAME(date)

功能：返回日期的星期名称。

例：

```
SELECT DAYNAME(DATE '2003-01-01');
```

查询结果为：Wednesday

13．函数 DAYOFMONTH

语法：DAYOFMONTH(date)

功能：返回日期为所处月份中的第几天。

例：

```
SELECT DAYOFMONTH('2003-01-03');
```

查询结果为：3

14．函数 DAYOFWEEK

语法：DAYOFWEEK(date)

功能：返回日期为所处星期中的第几天。

例：

```
SELECT DAYOFWEEK('2003-01-01');
```

查询结果为：4

15．函数 DAYOFYEAR

语法：DAYOFYEAR(date)

功能：返回日期为所处年中的第几天。

例：

SELECT DAYOFYEAR('2003-03-03');

查询结果为：62

16．DAYS_BETWEEN 函数

语法:　DAYS_BETWEEN(dt1,dt2)

功能：返回两个日期之间相差的天数。

例：

17．函数 EXTRACT

语法：EXTRACT(dtfield FROM date)

功能：EXTRACT 从日期时间类型或时间间隔类型的参数 date 中抽取 dtfield 对应的数值，并返回一个数字值。如果 date 是 NULL，则返回 NULL。Dtfiled 可以是 YEAR、MONTH、DAY、HOUR、MINUTE、SECOND。对于 SECOND 之外的任何域，函数返回整数，对于 SECOND 返回小数。

例：

SELECT EXTRACT(YEAR FROM DATE '2000-01-01');

查询结果为：2000

SELECT EXTRACT(DAY FROM DATE '2000-01-01');

查询结果为：1

SELECT EXTRACT(MINUTE FROM TIME '12:00:01.35');

查询结果为：0

SELECT EXTRACT(SECOND FROM TIMESTAMP '2000-01-01 12:00:01.35');

查询结果为：1.35

SELECT EXTRACT(SECOND FROM INTERVAL '-05:01:22.01' HOUR TO SECOND);

查询结果为：-22.01

18．函数 GETDATE

语法：GETDATE()

功能：返回系统的当前时间戳。

例：

SELECT GETDATE();

查询结果为：返回系统的当前日期时间，如 2003-02-27 16:22:24.0

19．函数 HOUR

语法：HOUR(time)

功能：返回时间中的小时分量。

例：

SELECT HOUR(TIME '20:10:16');

查询结果为：20

20．函数 LAST_DAY

语法：LAST_DAY(date)

功能：返回 date 所在月最后一天的日期,date 是日期类型(DATE)或时间戳类型(TIMESTAMP)，返回类型与 date 相同。

例：

```
SELECT LAST_DAY(SYSDATE) "Days Left";
```

查询结果为：如：当前日期为 2003 年 2 月的某一天，则结果为 2003-02-28

```
SELECT LAST_DAY(TIMESTAMP '2000-01-11 12:00:00');
```

查询结果为：2000-01-31

21．函数 MINUTE

语法：MINUTE(time)

功能：返回时间中的分钟分量。

例：

```
SELECT MINUTE('20:10:16');
```

查询结果为：10

22．函数 MONTH

语法：MONTH(date)

功能：返回日期中的月份分量。

例：

```
SELECT MONTH('2002-11-12');
```

查询结果为：11

23．函数 MONTHNAME

语法：MONTHNAME(date)

功能：返回日期中月份分量的名称。

例：

```
SELECT MONTHNAME('2002-11-12');
```

查询结果为：November

24．函数 MONTHS_BETWEEN

语法：MONTHS_BETWEEN(date1,date2)

功能：返回 date1 和 date2 之间的月份值。如果 date1 比 date2 晚，返回正值，否则返回负值。如果 date1 和 date2 这两个日期为同一天，或者都是所在月的最后一天，则返回整数，否则返回值带有小数。date1 和 date2 是日期类型(DATE)或时间戳类型(TIMESTAMP)。

例：

```
SELECT MONTHS_BETWEEN(DATE '1995-02-28', DATE '1995-01-31') "Months";
```

查询结果为：1.000000

```
SELECT MONTHS_BETWEEN(TIMESTAMP '1995-03-28 12:00:00',
TIMESTAMP '1995-01-31 12:00:00') "Months";
```

查询结果为：1.903226

25．函数 NEXT_DAY

语法：NEXT_DAY(date,char)

功能：返回在日期 date 之后满足由 char 给出的条件的第一天。char 指定了一周中的某一个天（星期几），返回值的时间分量与 date 相同，char 是大小写无关的。

Char 取值如下表所示：

星期描述说明

输入值	含义
SUN	星期日
SUNDAY	
MON	星期一
MONDAY	
TUES	星期二
TUESDAY	
WED	星期三
WEDNESDAY	
THURS	星期四
THURSDAY	
FRI	星期五
FRIDAY	
SAT	星期六
SATURDAY	

例：

```
SELECT NEXT_DAY(DATE '2001-08-02', 'MONDAY');
```

查询结果为：2001-08-06

```
SELECT NEXT_DAY('2001-08-02 12:00:00', 'FRI');
```

查询结果为：2001-08-03 12:00:00

26．函数 NOW

语法：NOW()

功能：返回系统的当前时间戳。等价于 GETDATE()。

27．函数 QUARTER

语法：QUARTER(date)

功能：返回日期在所处年中的季节数。

例：

```
SELECT QUARTER('2002-08-01');
```

查询结果为：3

28．函数 SECOND

语法：SECOND(time)

功能：返回时间中的秒分量。

例：

```
SELECT SECOND('08:10:25.300');
```

查询结果为：25

29．函数 ROUND

语法：ROUND(date[,format])

功能：将日期时间 date 四舍五入到最接近格式参数 format 指定的形式。如果没有指定语法的话，到今天正午 12P.M.为止的时间舍取为今天的日期，之后的时间舍取为第二天 12A.M.。日期时间 12A.M.，为一天的初始时刻。参数 date 的类型可以是 DATE 或 TIMESTAMP，但应与 format 相匹配。函数的返回结果的类型与参数 date 相同。format 具体如下表所示：

日期时间说明

format 的格式	含义	date 数据类型
cc, scc	世纪，从 1950、2050 等年份的一月一号午夜凌晨起的日期，舍取至下个世纪的一月一号	DATE TIMESTAMP
syear, syyy, y, yy, yyy, yyyy, year	年，从七月一号午夜凌晨起的日期，舍取至下个年度的一月一号	DATE TIMESTAMP
Q	季度，从十六号午夜凌晨舍取到季度的第二个月，忽略月中的天数	DATE TIMESTAMP
month, mm, m, rm	月，从十六号午夜凌晨舍取	DATE TIMESTAMP
Ww	舍取为与本年第一天星期数相同的最近的那一天	DATE TIMESTAMP
W	舍取为本月第一天星期数相同的最近的一天	DATE TIMESTAMP
ddd, dd, j	从正午起，舍取为下一天，默认值	DATE TIMESTAMP
day, dy, d	星期三正午起，舍取为下个星期天	DATE TIMESTAMP
hh, hh12, hh24	在一个小时的 30 分 30 秒之后的时间舍取为下一小时	TIME TIMESTAMP
Mi	在一个分钟 30 秒之后的时间舍取为下一分	TIME TIMESTAMP

有关 ww 和 w 的计算进一步解释如下（下面的时间仅当 date 参数为时间戳时才有效）：

ww 产生与本年第一天星期数相同的最近的日期。因为一个星期为七天，这意味着舍取结果在给定日期的三天半以内。例如，如果本年第一天为星期二，若给定日期在星期五上午 11:59:59 之前，则舍取为本星期的星期二的日期；否则舍取为下星期的星期二的日期。

w 计算的方式类似，不是产生最近的星期一 00:00:00，而是产生与本月第一天相同

的星期数的日期。

例：

```
SELECT ROUND(DATE '1992-10-27', 'scc');
```

查询结果为：2001-01-01

```
SELECT ROUND(DATE '1992-10-27', 'YEAR') "FIRST OF THE YEAR";
```

查询结果为：1993-01-01

```
SELECT ROUND(DATE '1992-10-27', 'q');
```

查询结果为：1992-10-01

```
SELECT ROUND(DATE '1992-10-27', 'month');
```

查询结果为：1992-11-01

```
SELECT ROUND(TIMESTAMP '1992-10-27 11:00:00', 'ww');
```

查询结果为：1992-10-28 00:00:00.0

```
SELECT ROUND(TIMESTAMP '1992-10-27 11:00:00', 'w');
```

查询结果为：1992-10-29 00:00:00.0

```
SELECT ROUND(TIMESTAMP '1992-10-27 12:00:01', 'ddd');
```

查询结果为：1992-10-28 00:00:00.0

```
SELECT ROUND(DATE '1992-10-27', 'day');
```

查询结果为：1992-10-25

```
SELECT ROUND(TIMESTAMP '1992-10-27 12:00:31', 'hh');
```

查询结果为：1992-10-27 12:00:00.0

```
SELECT ROUND(TIMESTAMP '1992-10-27 12:00:31', 'mi');
```

查询结果为：1992-10-27 12:01:00.0

30．函数 TIMESTAMPADD

```
语法：TIMESTAMPADD(interval,n,timestamp)
```

功能：返回时间戳 timestamp 加上 n 个 interval 代表的时间间隔的结果。interval 可以为 SQL_TSI_FRAC_SECOND、SQL_TSI_SECOND、SQL_TSI_MINUTE、SQL_TSI_HOUR、SQL_TSI_DAY、SQL_TSI_WEEK、SQL_TSI_MONTH、SQL_TSI_QUARTER 和 SQL_TSI_YEAR。

例：

```
SELECT TIMESTAMPADD(SQL_TSI_FRAC_SECOND, 5, '2003-02-10 08:12:20.300' );
```

查询结果为：2003-02-10 08:12:20.305

```
SELECT TIMESTAMPADD(SQL_TSI_YEAR, 30, DATE '2002-01-01');
```

查询结果为：2032-01-01 00:00:00.0

```
SELECT TIMESTAMPADD(SQL_TSI_QUARTER, 2, TIMESTAMP '2002-01-01 12:00:00');
```

查询结果为：2002-07-01 12:00:00.0

```
SELECT TIMESTAMPADD(SQL_TSI_DAY, 40, '2002-12-01 12:00:00');
```

查询结果为：2003-01-10 12:00:00.0

```
SELECT TIMESTAMPADD(SQL_TSI_WEEK, 1, '2002-01-30');
```

查询结果为：2003-02-06 00:00:00.0

31．函数 TIMESTAMPDIFF

语法：TIMESTAMPDIFF(interval,timestamp1,timestamp2)

功能：返回一个表明 timestamp2 与 timestamp1 之间的 interval 类型的时间间隔的整数。interval 可以为 SQL_TSI_FRAC_SECOND、SQL_TSI_SECOND、SQL_TSI_MINUTE、SQL_TSI_HOUR 、 SQL_TSI_DAY 、 SQL_TSI_WEEK 、 SQL_TSI_MONTH 、 SQL_TSI_QUARTER 和 SQL_TSI_YEAR。

注：当结果超出整数值范围，TIMESTAMPDIFF 产生错误。对于秒级 SQL_TSI_SECOND，最大数是 68 年。

例：

```
SELECT TIMESTAMPDIFF(SQL_TSI_FRAC_SECOND,
'2003-02-14 12:10:10.000', '2003-02-14 12:09:09.300');
```

查询结果为：-60700

```
SELECT TIMESTAMPDIFF(SQL_TSI_QUARTER, '2003-06-01', DATE '2002-01-01');
```

查询结果为：-5

```
SELECT TIMESTAMPDIFF(SQL_TSI_MONTH, '2001-06-01', DATE '2002-01-01');
```

查询结果为：7

```
SELECT TIMESTAMPDIFF(SQL_TSI_WEEK, DATE '2003-02-07',DATE '2003-02-14');
```

查询结果为：1

32．函数 SYSDATE

语法: SYSDATE()

功能： 获取系统当前时间。

例：

```
SELECT SYSDATE();
```

查询结果：当前系统时间

33．函数 TO_DATE

语法：TO_DATE (char [,fmt])

功能：将 CHAR 或者 VARCHAR 类型的值转换为 DATE 数据类型。其中 fmt 为指定字符串输出的日期语法，若省略 fmt，该函数使用缺省的日期语法。TO_DATE 中使用的有效的日期语法分量和 TO_CHAR 函数中的一致。

DM 缺省的日期语法为："YYYYMMDD HH:MI:SS"、"YYYY.MM.DD HH:MI:SS"、"YYYY-MM-DD　HH:MI:SS"或是"YYYY/MM/DD HH:MI:SS"。

合法的 DATE 格式为年月日、月日年和日月年三种，各部分之间可以有间隔符(".","-","/")或者没有间隔符；合法的 TIME 格式为：时分和时分秒,间隔符为":"。

例：

```
SELECT TO_DATE('2003-06-19 08:40:36','YYYY-MM-DD HH:MI:SS');
```

查询结果：2003-06-19 08:40:36.0

```
SELECT TO_DATE('2003/06/19','YYYY/MM/DD');
```
查询结果：2003-06-19 00:00:00.0
```
SELECT TO_DATE('20030619 08:40:36','YYYYMMDD HH24:MI:SS');
```
查询结果：2003-06-19 08:40:36.0

34．函数 TRUNC

语法：TRUNC(date[,format])

功能：将日期时间 date 截断到最接近格式参数 format 指定的形式。若 format 缺省，则返回当天日期。语法与 ROUND 类似，但结果是直接截断，而不是四舍五入。参数及函数的返回类型与 ROUND 相同。参见 ROUND。

例：

```
SELECT TRUNC(DATE '1992-10-27', 'scc');
```
查询结果为：1901-01-01
```
SELECT TRUNC(DATE '1992-10-27', 'YEAR') "FIRST OF THE YEAR";
```
查询结果为：1992-01-01
```
SELECT TRUNC(DATE '1992-10-27', 'q');
```
查询结果为：1992-10-01
```
SELECT TRUNC(DATE '1992-10-27', 'month');
```
查询结果为：1992-10-01
```
SELECT TRUNC(TIMESTAMP '1992-10-27 11:00:00', 'ww');
```
查询结果为：1992-10-21 00:00:00.0
```
SELECT TRUNC(TIMESTAMP '1992-10-27 11:00:00', 'w');
```
查询结果为：1992-10-22 00:00:00.0
```
SELECT TRUNC(TIMESTAMP '1992-10-27 12:00:01', 'ddd');
```
查询结果为：1992-10-27 00:00:00.0
```
SELECT TRUNC(DATE '1992-10-27', 'day');
```
查询结果为：1992-10-25
```
SELECT TRUNC(TIMESTAMP '1992-10-27 12:00:31', 'hh');
```
查询结果为：1992-10-27 12:00:00.00.0
```
SELECT TRUNC(TIMESTAMP '1992-10-27 12:00:31', 'mi');
```
查询结果为：1992-10-27 12:00:00.00.0

35．函数 WEEK

语法：WEEK(date)

功能：返回指定日期属于所在年中的第几周。

例：

```
SELECT WEEK(DATE '2003-02-10');
```
查询结果为：7

36．函数 WEEKDAY

语法：WEEKDAY(date)

功能：返回指定日期的星期值。如果是星期日则返回 0。

例：

```
SELECT WEEKDAY(DATE '1998-10-26');
```

查询结果：1

37．函数 WEEKS_BETWEEN

语法：WEEKS_BETWEEN(date1,date2)

功能：返回两个日期之间相差周数。

例：

```
SELECT WEEKS_BETWEEN(DATE '1998-2-28', DATE '1998-10-31');
```

查询结果：35

38．函数 YEAR

语法：YEAR(date)

功能：返回日期中的年分量。

例：

```
SELECT YEAR(DATE '2001-05-12');
```

查询结果为：2001

39．函数 YEARS_BETWEEN

语法：YEARS_BETWEEN(date1,date2)

功能：返回两个日期之间相差年数。

例：

```
SELECT    YEARS_BETWEEN(DATE '1998-2-28', DATE '1999-10-31');
```

查询结果为： 1

8.4　空值判断函数

空值判断函数用于判断参数是否为 NULL，或根据参数返回 NULL。

1．函数 COALESCE

语法：COALESCE(n1,n2,…,nx)

功能：返回其参数中第一个非空的值，如果所有参数均为 NULL，则返回 NULL。如果参数为多媒体数据类型，如 TEXT 类型，则系统会将 TEXT 类型先转换为 VARCHAR 类型或 VARBINARY 类型，转换的最大长度为 8188，超过部分将被截断。

例：

```
SELECT COALESCE(1,NULL);
```

查询结果：1

```
SELECT COALESCE(NULL,TIME '12:00:00',TIME '11:00:00');
```

查询结果：12:00:00

```
SELECT COALESCE(NULL,NULL,NULL,NULL);
```

查询结果：NULL

2．函数 IFNULL

语法：IFNULL(n1,n2)

功能：当表达式 n1 为非空时，返回 n1；若 n1 为空，则返回表达式 n2 的值。

例：

```
SELECT IFNULL(1,3);
```

查询结果：1

```
SELECT IFNULL(NULL,3);
```

查询结果：3

3．函数 ISNULL

语法：ISNULL(n1,n2)

功能：当表达式 n1 为非空时，返回 n1；若 n1 为空，则返回表达式 n2 的值。等价于 IFNULL()。

4．函数 NULLIF

语法：NULLIF(n1,n2)

功能：如果 n1=n2，返回 NULL，否则返回 n1。

例：

```
SELECT NULLIF(1,2);
```

查询结果：1

```
SELECT NULLIF(1,1);
```

查询结果：NULL

5．函数 NVL

语法：NVL(n1,n2)

功能：等价于 COALESCE(n1,n2)

8.5　类型转换函数

1．函数 CAST

语法：CAST(value AS type)

功能：将参数 value 转换为 type 类型返回。类型之间转换的相容性如下表所示：表中，“允许”表示这种语法有效且不受限制，“—”表示语法无效，“受限”表示转换还受到具体参数值的影响。

精确数值类型为：NUMERIC、DECIMAL、BYTE、INTEGER、SMALLINT。

近似数值类型为：FLOAT、REAL、DOUBLE PRECISION。

变长字符串为：VARCHAR，固定字符串为：CHAR，统称为字符串。

日期为：DATE。时间为：TIME。时间戳为：TIMESTAMP。

年月时间间隔为：INTERVAL YEAR TO MONTH、INTERVAL YEAR、INTERVAL MONTH。

日时时间间隔为：INTERVAL DAY、INTERVAL DAY TO HOUR、INTERVAL DAY TO MINUTE、INTERVAL DAY TO SECOND、INTERVAL HOUR、INTERVAL HOUR TO MINUTE、INTERVAL HOUR TO SECOND、INTERVAL MINUTE、INTERVAL MINUTE TO SECOND、INTERVAL SECOND。

MONEY 类型为以货币符号开头的数值类型。

多媒体数据类型不能转换。

CAST 类型转换相容矩阵

Value 数据类型	type 数据类型									
	精确数值	近似数值	变长字符串	固定字符串	日期	时间	时间戳	年月时间间隔	日时时间间隔	MONEY
精确数值	允许	允许	允许	允许	—	—	—	—	—	允许
近似数值	允许	允许	允许	允许	—	—	—	—	—	允许
字符串	允许	允许	受限	受限	允许	允许	允许	允许	允许	允许
日期	—	—	允许	允许	允许	—	允许	—	—	允许
时间	—	—	允许	允许	—	允许	允许	—	—	—
时间戳	—	—	允许	允许	允许	允许	允许	—	—	—
年月时间间隔	—	—	允许	允许	—	—	—	允许	—	—
日时时间间隔	—	—	允许	允许	—	—	—	—	允许	—
MONEY	允许	允许	允许	允许	—	—	—	—	—	允许

例：

```
SELECT CAST(100.5678 AS NUMERIC(10,2));
查询结果：100.57
SELECT CAST(100.5678 AS VARCHAR(8));
查询结果：'100.5678'
SELECT CAST('100.5678' AS INTEGER);
查询结果：100
SELECT CAST(INTERVAL '01-01' YEAR TO MONTH AS char(50));
查询结果：INTERVAL '1-1' YEAR(9) TO MONTH
```

2．函数 CONVERT

```
语法：CONVERT(type,value)
```

功能：将参数 value 转换为 type 类型返回。其类型转换相容矩阵与函数 CAST()的相同。

例：

SELECT CONVERT(VARCHAR(8),100.5678);

查询结果：'100.5678'

SELECT CONVERT(INTEGER, '100.5678');

查询结果：100

SELECT CONVERT(CHAR(50),INTERVAL '100-5' YEAR(3) TO MONTH);

查询结果：'INTERVAL '100-5' YEAR(3) TO MONTH'

8.6　杂类函数

1．DECODE 函数

语法：DECODE(exp, search1, result1, … searchn, resultn,[default])

功能：查表译码，DECODE 函数将 exp 与 search1,search2, … searchn 相比较，如果等于 searchx，则返回 resultx，如果没有找到匹配项，则返回 default, 如果未定义 default，返回 NULL。

例：

SELECT DECODE(1, 1, 'A', 2, 'B');

查询结果为：'A'

SELECT DECODE(3, 1, 'A', 2, 'B');

查询结果为：NULL

SELECT DECODE(3, 1, 'A', 2, 'B', 'C');

查询结果为：'C'

8.7　系统函数

系统函数可用于查询系统相关信息，或对系统进行日常维护、优化。对于没有输入参数的系统函数，调用时可以省略括号。

1．函数 CUR_DATABASE

语法：CUR_DATABASE()

功能：返回数据库名。

例：设 DM 服务器上名为 SYSTEM 的数据库已启动，执行

SELECT CUR_DATABASE();

查询结果为：SYSTEM

2．函数 DBID

语法：DBID()

功能：返回当前数据库标识符。

例：

```
SELECT　DBID();
```

查询结果为所在的数据库标识符。

3．函数 EXTENT

语法：EXTENT()

功能：取得当前每个簇中的块数。

例：

```
SELECT　EXTENT();
```

查询结果为：返回在建立数据库时设定的簇大小（16 或 32）。

4．函数 PAGE

语法：PAGE()

功能：取得当前每个块的字节数。

例：

```
SELECT　PAGE();
```

查询结果为：返回在建立数据库时设定的块大小（4K、8K、16K 或 32K）。

5．函数 SESSID

语法：SESSID()

功能：取得当前会话 标识符。

例：

```
SELECT　SESSID();
```

查询结果为：当前会话的标识符，如 50987056。

6．函数 UID

语法：UID()

功能：返回当前用户 ID。

例：

```
SELECT　UID();
```

查询结果为：当前用户的 ID 号，如 50331648。

7．函数 USER

语法：USER（）

功能：返回当前用户名。

例：

```
SELECT　USER();
```

查询结果为：当前用户名，如 SYSDBA。

8．函数 TABLEDEF

语法：TABLEDEF(db, schema, table)

功能：返回数据库 db 中模式 schema 的基表 table 的建表语句。

例：在 SYSTEM 库中，用户 SYSDBA 建一张表：

```
create table t1(c1 int primary key, c2 char(6) default 'abc');
```

然后查询该表的建表语句：

```
SELECT TABLEDEF('SYSTEM','SYSDBA','T1');
```

查询结果为：

```
CREATE TABLE "T1" AT "SYSTEM" (
"C1" INTEGER NOT NULL,
"C2" CHAR(6) DEFAULT 'abc',
CONSTRAINT " INDEX33555438" PRIMARY KEY("C1")
);
```

9．函数 VSIZE

语法：VSIZE(n)

功能：返回 n 的核心内部表示的字节数。如果 n 为 NULL，则返回 NULL。

例：沿用上例中的表 T1，插入两行数据，进行 VSIZE 查询。

```
insert into t1(c1) values(1);
insert into t1 values(2, '12345');
SELECT DISTINCT VSIZE(c1),VSIZE(c2) FROM t1;
```

查询结果如下表 8.7.1 所示。

表 8. 7. 1 查询结果

VSIZE(c1)	VSIZE(c2)
4	6

10．函数 SET_TABLE_OPTION

语法：SET_TABLE_OPTION([db.][sch.]tablename, option, value)

功能：设置基表 tablename 的选项值。当应用程序能保证不对表进行增、删、改时，可以禁用表的页级锁和行级锁，从而提高查询速度。选项 option 可取值为 disallowpagelocks（指定对表的页级锁的上锁方式），或 disallowrowlocks（指定对表的行级锁的上锁方式）。value 取值为 0 或 1，1 表示不上锁，0 表示上锁。

例：某应用中，对表 tt 不做修改，则可以使用下列语句禁止对表 tt 的页锁和行锁：

```
select set_table_option('tt','disallowpagelocks',1);
select set_table_option('tt','disallowrowlocks',1);
```

此时，表 tt 的并发度是最高的。

11．函数 SET_INDEX_OPTION

语法：SET_INDEX_OPTION([[db.].schema.]indexname, option, value)

功能：设置索引 indexname 的选项值。选项 option 可取值为 disallowpagelocks 或 disallowhashsearch，value 取值为 0 或 1，参数意义如下。

option 选项名	选项意义	选项值 value 意义
disallowpagelocks	指定对索引的页级锁的上锁方式	1：不上锁 / 0：上锁
disallowhashsearch	指定对索引的搜索方式	1：快速搜索/ 0：HASH 搜索

例：某应用中，对索引 ind1 不做修改，则可以使用下列语句禁止对它的页锁：

```
select set_index_option('ind1','disallowpagelocks',1);
```

此时，索引 ind1 的并发度是最高的。

12. 函数 UNLOCK_USER

语法：UNLOCK_USER (LoginName)

功能：解除对登录 LoginName 的登录锁定。返回值为 0 表示解锁失败，1 表示成功。

例：假设用户使用登录 MYLOGIN 多次登录失败，被系统锁定。SYSDBA 有权帮其

解锁：

```
select unlock_user ('MYLOGIN');
```

这样，登录 MYLOGIN 被激活，可以重新使用。

13．函数 GET_AUDIT

语法：GET_AUDIT()

功能：查询当前系统的审计开关状态。

例：查询当前审计开关状态：

```
select get_audit();
```

返回值为 0 表示审计开关已打开，若为 1 表示关闭。

14．函数 SET_AUDIT

语法：SET_AUDIT(n)

功能：设置系统的审计开关状态，n 可以取值为 0 或 1。0 表示打开审计开关，1 表示关闭审计开关。

例：打开系统审计开关：

```
select set_audit(0);
```

返回值为 0 表示审计开关已打开，若为 1 则表示审计开关已关闭，否则表示设置失败。

第 9 章　数据库元信息

数据字典是整个 DM 数据库的信息中心和情报系统，是数据库中的数据库。数据字典由一系列系统表组成，只不过这些表不是由用户建立而是由系统建立的，我们称之为系统表，其表名均以“SYS”开头。当一个管理信息系统建成后，数据字典就是数据库各种用户观察数据库的窗口。用户需要经常阅读这些表的内容，了解系统性能、空间使用情况及各种统计信息，以便及时掌握数据库的运行状态，有效控制系统高效地运行。为此，DM 提供了数据字典查询语句，可供数据库各种用户在规定的权限范围内查询数据字典的有关内容，为用户随时了解和掌握系统工作情况提供了方便，也为用户在运行和调试应用程序过程中及时了解系统状态提供了方便。

DM 参照 SQL92 对信息模式进行了比较全面的支持，DM 的信息模式由系统自动创建，且自动授查询权限给每个用户，用户即使不熟悉 DM 的字典结构，也可以方便地查询信息模式中的表和视图，从而了解整个数据库的字典信息。

9.1　数据字典查询语句

DM 系统提供的数据字典查询语句，可供系统管理员和一般用户在规定的权限范围内利用交互方式查询数据字典中的有关内容。

数据字典查询的语法描述为：

语法格式

```
SELECT  * | <列名>[,<列名>...]
            FROM   <字典>[,<字典>...]
            [ WHERE  <检索条件> ]
            [ ORDER  BY  <排序说明> [ {, <排序说明> } ] ]

<字典> ::= <全局字典>|<局部字典>
<全局字典>::=[[SYSTEM.]SYSDBA.]<普通全局字典> |
            [[SYSTEM.]SYSAUDITOR.]<审计全局字典>
<普通全局字典>::=SYSDATABASES| SYSFILES| SYSLOGINS| SYSPWDCHGS|
                SYSPRIVILEGES| SYSSTATISTICS| SYSAUTHS|
                SYSTYPEINFOS|SYSEVENTS|SYSACTORS|SYSDUPLICATES|
                SYSDUPLICATESLAVES|SYSBACKUPS| SYSCTLFILES|
                SYSLOGFILES| SYSDEVICES
<审计全局字典>::=SYSAUDIT| SYSAUDITRECORDS
<局部字典>::=[[数据库名.]SYSDBA.]<局部字典表名>
<局部字典表名>::=SYSTABLES| SYSCOLUMNS | SYSINDEXES| SYSINDEXKEYS|
             SYSVIEWS| SYSDEPCOLS| SYSPROCS| SYSARGS| SYSDEPENDS|
             SYSTRIGGERS| SYSCOLTRIGGERS| SYSCONSTRAINTS| SYSCHECKS|
```

```
SYSCHKCONSTRAINTS| SYSREFCONSTRAINTS| SYSIDENTITY|
SYSDUAL| SYSSEQUENCES| SYSUSERS| SYSROLES| SYSSCHEMAS|
SYSSYSGRANTS| SYSTVGRANTS| SYSCOLGRANTS| SYSPFGRANTS|
SYSURGRANTS| SYSCONTEXTINDEXES
```

语法规则

字典查询与一般查询语法一致，具体语法规则请参见第 4 章。

注意：用户对系统表只能查询，不能更改其结构和数据。

9.2 数据字典表结构

DM 系统提供的字典表总计 53 张。所有系统管理员和所有一般用户可以做下面的查询从字典表 SYSTABLES 得到所有字典表的统计信息。查询语句如下：

```
SELECT * FROM SYSTEM.SYSDBA.SYSTABLES WHERE TYPE = 'S' AND NAME like 'SYS%';
```

在交互方式下，数据字典查询语句的查询结果由系统自动以表格形式在屏幕上显示。

关于 DM 系统中提供的所有系统表中包含的列及每列的类型、长度、含义等具体介绍参见《DM 系统管理员手册》3.5 节。

以下系统表仅供系统内部使用，用户不需查看:

SYSPROCSDMBLOB

SYSTMPDMBLOB

SYSTRIGGERSDMBLOB

SYSVIEWSDMBLOB

9.3 信息模式

DM 数据库对信息模式的实现采用在系统表上建立视图来实现，这些表和视图都由系统自动建立，用户只能进行查询，而无更新和删除的权限。

1．信息模式 INFORMATION_SCHEMA 与 INFO_SCHEM 的对应关系

DM 支持 2 个信息模式 INFORMATION_SCHEMA 和 INFO_SCHEM。

因为 INFO_SCHEM 是 NIST 采纳了 ANSI/X3 技术委员会的下述建议之后纳入测试范围的：鉴于 FIPS 127-2 标准中有一个矛盾的规定，即在中级之前的级别中并不需要提供长度超过 18 的列名的支持，而过渡级支持的 INFORMATION_SCHEMA 中有长度超过 18 的列名。为了解决此问题，该委员会建议对 FIPS 过渡级 SQL 进行修改以支持一个列名比 INFORMATION_SCHEMA 的列名更短的特殊模式。

该委员会还提出了对一些关键字的缩写：

DEFAULT ---- DEF

CHARACTER ---- CHAR

MAXIMUM ---- MAX

PRECISION ---- PREC
CATALOG ---- CAT
SCHEMA ---- SCHEM
NUMERIC ---- NUM

由此可见，INFO_SCHEM 是对 INFORMATION_SCHEMA 命名的一种简化，所以主要解释 INFORMATION_SCHEMA 的基表和视图的含义。

DM 支持信息模式的一个基表 INFORMATION_SCHEMA_CATALOG_NAME 和 12 个视图：SCHEMATA、TABLES、VIEWS、COLUMNS、TABLE_PRIVILEGES、COLUMN_PRIVILEGES、TABLE_CONSTRAINTS、REFERENTIAL_CONSTRAINTS、CHECK_CONSTRAINTS、KEY_COLUMN_USAGE、VIEW_TABLE_USAGE、USAGE_PRIVILEGES。

各视图的具体情况介绍参见《DM 系统管理员手册》3.6 节。

第 10 章　一致性和并发性

数据一致性是指表示客观世界同一事物状态的数据，不管出现在何时何处都是一致的、正确的和完整的。数据库是一个共享资源，可为多个应用程序所共享，它们同时存取数据库中的数据，这就是数据库的并发操作。此时，如果不对并发操作进行控制，则会存取不正确的数据，或破坏数据库数据的一致性。

DM 利用事务和封锁机制提供数据并发存取和数据完整性。在一事务内由语句获取的全部封锁在事务期间被保持，防止其它并行事务的破坏性干扰。一个事务的 SQL 语句所做的修改在它提交后才可能为其它事务所见。

DM 自动维护数据库的一致性和完整性，并允许选择实施事务级读一致性，它保证同一事务内的可重复读，为此 DM 提供用户各种手动上锁语句和设置事务隔离级别语句。

本章介绍 DM 中和事务管理相关的 SQL 语句和手动上锁语句。在本章各例中，**如不特别说明，各例的当前数据库均为 CW，用户均为建表者 USER1，因此表名、视图名前通常省略了数据库名、模式名前缀。**

10.1　DM 事务相关语句

DM 中事务是一个逻辑工作单元，由一系列 SQL 语句组成。DM 把一个事务的所有 SQL 语句作为一个整体，即事务中的操作，要么全部执行，要么一个也不执行。

10.1.1　事务的开始

DM 没有提供显式定义事务开始的语句，第一个可执行的 SQL 语句(除登录语句外)隐含事务的开始。

10.1.2　事务的结束

用户可以使用显式的提交或回滚语句来结束一个事务，也可以隐式的结束一个事务。

1. 提交语句

语法格式

```
COMMIT [WORK]; |
COMMITWORK;
```

参数

WORK　　支持与标准 SQL 的兼容性，COMMIT 和 COMMIT WORK 等价。

功能

该语句使当前事务工作单元中的所有操作“永久化”，并结束该事务。

举例说明

例 插入数据到表厂商登记并提交。

```
INSERT INTO 厂商登记(厂商编号,厂商名,厂商地址)
        VALUES('B0007', '幸福公司', '北京');
COMMIT WORK;
```

2．回滚语句

语法格式

```
ROLLBACK [WORK]; |
```

功能

该语句回滚(废除)当前事务工作单元中的所有操作，并结束该事务。

使用说明

建议用户退出时，用 COMMIT 或 ROLLBACK 命令来显式地结束应用程序。如果没有显式地提交事务，而应用程序又非正常终止，则最后一个未提交的工作单元被回滚。

举例说明

例 插入数据到表厂商登记后回滚。

```
（1）往表厂商登记中插入一个数据
    INSERT INTO 厂商登记(厂商编号,厂商名,厂商地址)
    VALUES('B0007', '幸福公司', '北京');
（2）查询表厂商登记
    SELECT * FROM 厂商登记;
    /*厂商名为'幸福公司'的记录已经插入到表厂商登记中*/
（3）回滚插入操作
    ROLLBACK WORK;
（4）查询表厂商登记
    SELECT * FROM 厂商登记;
    /*插入操作被回滚，表厂商登记中不存在厂商名为'幸福公司'的记录*/
```

3．隐式的事务结束

事务开始后，当遇到下列情况时，该事务结束。该用户的下一个可执行的 SQL 语句将开始下一个新的事务。

（1）当一连接的属性设置为自动提交，每执行一条语句都会提交。

（2）在非自动提交事务中，当 DDL_AUTO_COMMIT 开关打开时，表示遇到任一 DDL 语句系统自动提交该 DDL 语句及之前的 DML 和 DDL 操作；当该开关关闭时，只有 CREATE DATABASE 和 ALTER DATABASE 两种 DDL 语句以及之前的 DML 和 DDL 操作会自动提交。

（3）事务所在的程序正常结束和用户退出。

（4）系统非正常终止时。

10.1.3 保存点相关语句

SAVEPOINT 语句用于在事务中设置保存点。保存点提供了一种灵活的回滚，事务

在执行中可以回滚到某个保存点，在该保存点以前的操作有效，而以后的操作被回滚掉。一个事务中可以设置多个保存点。

1．设置保存点

语法格式

```
SAVEPOINT <保存点名>;
```

参数

<保存点名>　　指明保存点的名字。

使用说明

一个事务中可以设置多个保存点，但不能重名。

2．回滚到保存点

语法格式

```
ROLLBACK [WORK] TO SAVEPOINT <保存点名>;
```

参数

（1）WORK　　支持与标准 SQL 的兼容性，ROLLBACK 和 ROLLBACK WORK 等价。

（2）<保存点名> 指明部分回滚时要回滚到的保存点的名字。

使用说明

回滚到保存点后事务状态和设置保存点时事务的状态一致，在保存点以后对数据库的操作被回滚。

举例说明

例 插入数据到表厂商登记后设置保存点，然后再插入另一数据，回滚到保存点。

```
（1）往表厂商登记中插入一个数据
    INSERT INTO 厂商登记(厂商编号,厂商名,厂商地址)
    VALUES('B0007', '幸福公司', '北京');
（2）查询表厂商登记
    SELECT * FROM 厂商登记;
    /*厂商名为'幸福公司'的记录已经被插入到表厂商登记中*/
（3）设置保存点
    SAVEPOINT A;
（4）往表厂商登记中插入另一个数据
    INSERT INTO 厂商登记(厂商编号,厂商名,厂商地址)
    VALUES('B0C05', '上海第一百货', '上海');
（5）回滚到保存点
    ROLLBACK TO SAVEPOINT A;
（6）查询表厂商登记
    SELECT * FROM 厂商登记;
    /*插入操作被回滚，厂商登记中不存在厂商名为'上海第一百货'的记录*/
```

10.1.4　设置事务隔离级及读写特性

事务的隔离级描述了给定事务的行为对其它并发执行事务的暴露程度。通过选择四个隔离级中的一个，用户能增加对其它未提交事务的暴露程度，获得更高的并发度。DM 允许用户改变未启动的事务的隔离级和读写特性，即下列语句必须在事务开始时执行，否则无效。

1．设置事务隔离级语句

事务的隔离级描述了给定事务的行为对其它并发执行事务的暴露程度。通过选择四个隔离级中的一个，用户能增加对其它未提交事务的暴露程度，获得更高的并发度。

语法格式

```
SET TRANSACTION ISOLATION LEVEL
    [READ UNCOMMITTED |
    READ COMMITTED    |
    REPEATABLE READ   |
    SERIALIZABLE ];
```

使用说明

（1）该语句设置事务的隔离级别（从上到下，隔离级别依次升高）：

----脏读（READ UNCOMMITTED）：事务隔离的最低级别，事务可能查询到其它事务未提交的数据，仅可保证不读取物理损坏的数据。

----读提交（READ COMMITTED）：DM 默认级别，保证不读脏数据。

----可重复读（REPEATABLE READ）：保证可重复读，但有可能读入幻像数据。

----可串行化（SERIALIZABLE）：事务隔离的最高级别，事务之间完全隔离。

一般情况下，使用读提交隔离级别可以满足大多数应用，如果应用要求可重复读以保证基于查询结果的更新的正确性就必须使用可重复读或可串行读隔离级别。

（2）只能在事务未开始执行前设置隔离级，事务执行期间不能更改隔离级。

2．设置事务读属性的语句!

语法格式

```
SET TRANSACTION READ ONLY | WRITE;
```

参数

（1）READ ONLY　　只读事务，该事务只能做查询操作，不能更新数据库。

（2）READ WRITE　　读写事务，该事务可以查询并更新数据库，是 DM 的默认设置。

功能

该语句设置事务的读写属性。

10.2　DM 手动上锁语句

DM 的隐式封锁足以保证数据的一致性，但用户可以根据自己的需要手工显式锁定表，允许或禁止在当前用户操作期间其它用户对此表的存取。DM 提供给用户四种表锁的封锁：意向共享锁（IS）、共享锁（S）、意向排它锁（IX）和排它锁（X）。

语法格式

```
LOCK TABLE [[<数据库名>.]<模式名>.]<表名> IN <封锁方式> MODE;

<封锁方式>::=
    INTENT SHARE |
    INTENT EXCLUSIVE |
    SHARE |
    EXCLUSIVE
```

使用说明

1．意向共享表封锁：INTENT SHARE TABLE LOCKS (IS)

该封锁表明该事务封锁了表上的一些元组并试图修改它们（但是还未做修改，其它事务可读这些元组，但是不能修改这些元组）。意向共享表封锁是限制最少的锁，提供了表上最大的并发度。

（1）允许操作：

其他事务对该表的并发查询、插入、更新、删除或在该表上进行封锁，其他事务可以同时上意向共享锁（S）、意向排他锁(IX)、共享锁(S)。

（2）禁止操作：

其它事务以排它方式(X)存取该表。

LOCK TABLE tablename IN EXCLUSIVE MODE;

2．意向排它表封锁：INETENSIVE EXCLUSIVE TABLE LOCKS (IX)

该锁表明该事务对表的元组进行一次或多次修改(其它事务不能访问这些元组)，行排它表封锁较行共享表封锁稍严格。

（1）允许操作：

其它事务并行查询、插入、更新、删除或封锁该表上行，允许多个事务在同一表上获得意向排它（IX）和意向共享(IS)表锁。

（2）禁止操作：

其它事务对表进行共享或者排它读写封锁：S, X

LOCK TABLE tablename IN SHARE MODE;

LOCK TABLE tablename IN EXCLUSIVE MODE;

3．共享表封锁：SHARE TABLE LOCKS (S)

该锁表明该事务访问表中所有元组，其他事务不能对该表做任何更新操作

（1）允许操作：

其它事务在该表上作查询，但是不允许作修改，且允许多个事务在同一表上并发地持有共享表封锁。

（2）禁止操作：

其它事务执行：

LOCK TABLE tablename IN INTENT EXCLUSIVE MODE;

LOCK TABLE tablename IN EXCLUSIVE MODE;

4．排它表封锁：EXCLUSIVE TABLE LOCKS (X)

该封锁是表封锁中最严格的方式，只允许持有封锁的事务可对该表进行修改

（1）允许操作：

不允许任何操作

（2）禁止操作：

其它事务对表执行任何 DML 语句，即不能插入、修改和删除该表中的行，封锁该表中的行或以任何方式封锁表。

5．用户上锁成功后锁将一直有效，直到当前事务结束时，该锁被系统自动解除。

举例说明

例 当用户 RU 希望独占某表，他可以对该表显式地上排它锁。

```
LOCK TABLE CW.USER1.厂商登记 IN EXCLUSIVE MODE;
```

第 11 章　存储模块

DM 系统允许用户使用 DM 提供的 DMPL/SQL 语言创建过程或函数，这些过程或函数象普通的过程或函数一样，有输入、输出参数和返回值，它们与表和视图等数据库对象一样被存储在数据库中，供用户随时调用。存储过程和存储函数在功能上相当于客户端的一段 SQL 批处理程序，但是在许多方面有着后者无法比拟的优点，它为用户提供了一种高效率的编程手段，成为现代数据库系统的重要特征。通常，我们将存储过程和存储函数统称为存储模块。

DM 的存储模块机制是一种技术，而不是一种独立的工具，它是和服务器紧密结合在一起的。可以认为这种技术是执行 DMPL/SQL 语言的一种机器，它可以接受任何有效的存储模块，按照语言本身所规定的语义执行，并将结果返回给客户。

DM 的存储模块机制具有如下优点：

1．提供更高的生产率：在设计应用时，围绕存储过程/函数设计应用，可以避免重复编码，提高生产率；在自顶向下设计应用时，不必关心实现的细节；编程方便。

2．便于维护：用户的存储模块在数据库集中存放，用户可以随时查询、删除、修改它们，而应用程序可以不作任何修改，或只做少量调整。

3．提供更好的性能：

（1）存储模块在创建时被编译成伪码序列，在运行时不需要重新进行编译和优化处理，它具有更快的执行速度，可以同时被多个用户调用，并能够减少操作错误；

（2）存储模块在执行时数据对用户是不可见的，提高了数据库的安全性；

（3）存储模块具有更高的可靠性；

（4）存储模块是一种高效访问数据库的机制，使用存储模块可减少应用对 DM 的调用，降低了系统资源浪费，显著提高性能，尤其是在网络上与 DM 通讯的应用更显著。

在本章各例中，**如不特别说明，各例的当前数据库均为 CW，用户均为建表、建存储过程、建存储函数者 USER1，因此表名、存储过程名、存储函数名前通常省略了数据库名、模式名前缀。**

11.1　存储模块的定义

本节中，我们将介绍如何使用 DMPL/SQL 语言来定义一个存储模块。为使用户获得对存储模块的整体印象，本节中提供了一些例子，对于例子中涉及到的各种控制语句，其详细的使用方法，请参阅 11.3 节“存储模块的控制语句”。

定义一个存储模块的详细语法如下：

```
<存储模块定义语句> ::=
<存储过程定义语句> |
<存储函数定义语句>

<存储过程定义语句> ::= CREATE [OR REPLACE] <过程>

<过程> ::= <过程申明> AS | IS <模块体>

<过程申明> ::= PROCEDURE <过程名定义> [ (<参数列>)]
<过程名定义> ::=[ <模式名>.]<过程名>

<存储函数定义语句> ::= CREATE [OR REPLACE] <函数>
```

```
<函数> ::= <函数申明> AS | IS <模块体>

<函数申明> ::= FUNCTION <函数名定义> [(<参数列>)] RETURN <返回数据类型>
<函数名定义> ::=[ <模式名>.]<函数名>
<参数列> ::= <参数申明>[{, <参数申明>}…]

<参数申明> ::= <参数名> [<参数模式>] <参数类型> [:= | DEFAULT <表达式>]

<参数模式> ::= IN | OUT | IN OUT

<模块体> ::=
        [<说明部分>]
        <执行部分>
        [<异常处理部分>]
        END

<说明部分> ::=
        [{<变量说明> | <异常变量说明> | <游标定义>;}…]

<变量说明> ::= <变量名> <变量类型> [:= | DEFAULT <表达式>]

<异常变量说明> ::= <异常变量名> EXCEPTION [FOR <错误号>]

<游标变量说明> ::= <游标变量名> CURSOR

<游标定义> ::= <游标名> CURSOR FOR <查询语句>

<执行部分> ::= BEGIN < SQL 过程语句序列>

< SQL 过程语句序列> ::= { [<标号说明>…] <SQL 过程语句>;}

<标号说明> ::= << <标号名> >>

<SQL 过程语句> ::= <SQL 语句> | <SQL 控制语句>

<异常处理部分> ::= EXCEPTION {<异常处理语句>;}…

<异常处理语句> ::= WHEN <异常名> THEN < SQL 过程语句序列>
```

1、存储过程

定义一个存储过程，应使用下面的语句：

语法格式

```
CREATE [OR REPLACE ] PROCEDURE <存储过程名定义>
      [(<参数名> <参数模式> <参数类型> {,<参数名> <参数模式> <参数类型>})]
```

```
        [WITH    ENCRYPTION]
        AS | IS
        [<说明部分>]
        <执行部分>
        [<异常处理部分>]
        END;
    <存储过程名定义> ::=[<模式名>.] <存储过程名> [AT <数据库名>]
```

参数

（1）<存储过程名>　指明被创建的存储过程的名字。

（2）<数据库名>　指明被创建的存储过程所属数据库的名字，缺省为当前数据库名。

（3）<模式名>　指明被创建的存储过程所属模式的名字，缺省为当前模式名。

（4）<参数名>　指明存储过程参数的名称。

（5）<参数模式>　参数模式可设置为 IN、OUT 或 IN OUT，缺省为 IN 类型。

（6）<参数类型>　指明存储过程参数的数据类型。

（7）<说明部分>　由变量、游标和子程序等对象的申明构成，可缺省。

（8）<执行部分>　由 SQL 语句和过程控制语句构成的执行代码。

（9）<异常处理部分>各种异常的处理程序，存储过程执行异常时调用，可缺省。

使用说明

（1）DM_SQL 过程语言支持的 SQL 语句包括：部分数据定义语句（CREATE、DROP 临时表），数据查询语句（SELECT），数据操纵语句（INSERT、DELETE、UPDATE），游标定义及操纵语句（DECLARE CURSOR、OPEN、FETCH、CLOSE），事务控制语句（COMMIT、ROLLBACK），动态 SQL 执行语句（EXECUTE IMMEDIATE）。

（2）DM_SQL 过程控制语句包括：语句块、赋值语句、IF 语句、LOOP 语句、WHILE 语句、FOR 语句、EXIT 语句、调用语句、RETURN 语句、NULL 语句、GOTO 语句、RAISE 语句和打印语句等。

（3）存储过程中的 RETURN 语句不允许带返回值。

（4）在定义存储过程时使用赋值符号‘:=’或关键字 DEFAULT，可以为 IN 参数指定一个缺省值。拥有缺省参数的参数必须在参数列表的尾端。

权限

使用该语句的用户必须具有 CREATE PROCEDURE 系统权限。

举例说明

例 在数据库 CW 下，用户 USER1 定义了一个简单的存储过程：

```
    CREATE OR REPLACE PROCEDURE USER1.PROC_1(A IN OUT INT)    AS
        B    INT;
    BEGIN
        A:=A+B;
 EXCEPTION
    WHEN OTHERS THEN NULL;
    END;
```

第 2 行是该存储过程的说明部分，这里申明了一个变量 B。注意在 DMPL/SQL 语言中说明变量时，变量的类型放在变量名称之后。第 3 行和第 4 行是该程序块运行时被执行的代码段，这里将 A 与 B 的和赋给参数 A。如果发生了异常，第 5 行开始的异常处理部分就对产生的异常情况进行处理，WHEN OTHERS 异常处理器处理所有不被其他异常处理器处理的异常，它必须是最后一个异常处理器。第 11.5 节详细介绍了异常的处理过程。说明部分和异常处理部分都是可选的。如果用户在模块中不需要任何局部变量或者不想处理发生的异常，则可以省略这两部分。

2、存储函数

存储函数和存储过程很相似，它们的区别在于：

（1）存储过程没有返回值，而存储函数有；

（2）存储过程中可以没有返回语句，而存储函数必须通过返回语句结束；

（3）存储过程的返回语句中不能带表达式，而存储函数必须带表达式；

（4）存储过程不能出现在一个表达式中，而存储函数只能出现在表达式中。

定义一个存储函数，应使用下面的语句：

语法格式

```
CREATE [OR REPLACE ] FUNCTION <存储函数名定义>
    [(<参数名> <参数模式> <参数类型>{,<参数名> <参数模式> <参数类型>})]
    RETURN <返回数据类型>
    [WITH    ENCRYPTION]
    AS | IS
    [<说明部分>]
    <执行部分>
    [<异常处理部分>]
    END;

<存储函数名定义> ::=[<模式名>.] <函数名> [AT <数据库名>] |
    [[<数据库名>.] <模式名>.]<函数名>
```

参数

（1）<存储函数名>　　指明被创建的存储函数的名字。

（2）<数据库名>　　指明被创建的存储函数所属数据库的名字，缺省为当前数据库名。

（3）<模式名>　　指明被创建的存储函数所属模式的名字，缺省为当前模式名。

（4）<参数名>　　指明存储函数参数的名称。

（5）<参数模式>　　参数模式可设置为 IN、OUT 或 IN OUT，缺省为 IN 类型。

（6）<参数类型>　　指明存储函数参数的数据类型。

（7）<返回数据类型> 指明存储函数返回值的数据类型。

使用说明

（1）DM_SQL 过程语言支持的 SQL 语句包括：部分数据定义语句（CREATE、DROP 临时表），数据查询语句（SELECT），数据操纵语句（INSERT、DELETE、UPDATE），游标定义及操纵语句（DECLARE CURSOR、OPEN、FETCH、CLOSE），事务控制语句（COMMIT、ROLLBACK），动态 SQL 执行语句（EXECUTE IMMEDIATE）。

（2）DM_SQL 过程控制语句包括：语句块、赋值语句、IF 语句、LOOP 语句、WHILE 语句、FOR 语句、EXIT 语句、调用语句、RETURN 语句、NULL 语句、GOTO 语句、RAISE 语句和打印语句等。

（3）在存储函数中必须使用 RETURN 语句向函数的调用环境返回一个值。

（4）存储函数不能用 CALL 语句调用，它只能出现在表达式中。

权限

使用该语句的用户必须具有 CREATE FUNCTION 系统权限。

举例说明

例　在数据库 CW 下，用户 USER1 定义一个简单的存储函数：

```
 CREATE OR REPLACE FUNCTION USER1.FUN_1(A INT, B INT) AT CW RETURN INT AS
    S    INT;
BEGIN
    S:=A+B;
```

```
        RETURN S;
  EXCEPTION
    WHEN OTHERS THEN NULL;
  END;
```

第 1 行说明了该函数的返回类型为 INT 类型。第 4 行将两个参数 A、B 的和赋给了变量 S，第 5 行的 RETURN 语句则将变量 S 的值作为函数的返回值返回。

3、参数

存储模块及模块中定义的子模块都可以带参数，用来给模块传送数据及向外界返回数据。在过程或函数中定义一个参数时，必须说明名称、参数模式和数据类型。三种可能的参数模式是，IN（缺省模式）、OUT 和 IN OUT，意义分别为：

（1）IN：　输入参数，用来将数据传送给模块；

（2）OUT：输出参数，用来从模块返回数据到进行调用的模块；

（3）IN OUT：既作为输入参数，也作为输出参数。

在存储模块中使用参数时要注意下面几点：

（1）IN 参数不能被赋值；

（2）OUT 参数的初值始终为空，无论调用该模块时对应的实参值为多少；

（3）调用一个模块时，OUT 参数及 IN OUT 参数的实参必须是可赋值的对象。

请看下面的例子：

```
CREATE OR REPLACE PROCEDURE PROC_ARG(A IN INT, B INT) AS
      BEGIN
         A:=0;        /*  错误：A 是 IN 参数，不能作为赋值对象  */
         B:=A+1;      /*  错误：B 是 IN 参数，不能作为赋值对象  */
      END;
```

这个存储过程是错误的，因为在第 3 行和第 4 行的语句中企图给 IN 参数赋值。注意，参数 B 虽然没有明确指定模式，但是其缺省为 IN 模式。

使用赋值符号‘:=’或关键字 DEFAULT，可以为 IN 参数指定一个缺省值。如果调用时未指定参数值，系统将自动使用该参数的缺省值。例如：

```
CREATE PROCEDURE PROC_DEF_ARG(A VARCHAR(10) DEFAULT 'ABC', B INT:=123) AS
BEGIN
  PRINT A
  PRINT B
END;
```

调用过程，不指定输入参数值：

```
CALL PROC_DEF_ARG;
```

系统使用缺省值作为参数值，打印结果为：

ABC

123

4、变量

变量的申明应在说明部分。使用赋值符号‘:=’或关键字 DEFAULT，可以为变量指定一个缺省值。其语法为：

语法格式

```
变量名 <变量类型> { := | DEFAULT}<表达式>]
```

举例说明

例　下例中说明了各种类型的变量：

```
CREATE OR REPLACE PROCEDURE PROC_VAR1 AS
   V1      VARCHAR(20);
   V2      INT:=25;
   V3      DATE DEFAULT DATE '2000-01-01';
BEGIN
   NULL;
END;
```

第 2 行说明了一个 VARCHAR 类型的变量。第 3 行说明了一个 INT 类型的变量，并将它的缺省值置为 25。第 4 行说明了一个 DATE 类型的变量，并将它的缺省值置为'2000-01-01'，。

变量的作用域与 PASCAL 语言类似。它只在定义它的语句块（包括其下层的语句块）内可见，并且定义在下一层语句块中的变量可以屏蔽上一层的同名变量。当遇到一个变量名时，系统首先在当前语句块内查找变量的定义；如果没有找到，再向包含该语句块的上一层语句块中查找，如此直到最外层。

下例中，存储过程 PROC_VAR2 中定义了一个整型变量 A，而在其下层的语句块中又定义了一个同名的字符型变量 A。由于在该语句块中，字符型变量 A 屏蔽了整型变量 A，所以第一条打印语句打印的是字符型变量 A 的值，而第二条打印语句打印的则是整型变量 A 的值。例如：

```
CREATE OR REPLACE PROCEDURE USER1.PROC_VAR2 AS
         A INTEGER :=5;
      BEGIN
           DECLARE
           A VARCHAR(10);        /*  此处定义的变量 A 与上一层中的变量 A 同名  */
        BEGIN
           A:= 'ABCDEFG';
           PRINT A;              /*  第一条打印语句  */
        END;
           PRINT A;              /*  第二条打印语句  */
      END;
```

执行此过程：

```
CALL    CW.USER1.PROC_VAR2;
```

打印结果应为：

ABCDEFG

5

5、使用 OR REPLACE 选项

读者可能已经发现，在前面的例子中，我们在创建存储模块的时候，都使用了 OR REPLACE 选项。使用 OR REPLACE 选项的好处是，如果系统中已经有同名的存储模块，服务器会自动用新的代码覆盖原来的代码。如果不使用 OR REPLACE 选项，当创建的存储模块与系统中已有的存储模块同名时，服务器会报错。

11.2　存储模块的删除

当用户需要从数据库中删除一个存储模块时，可以使用存储模块删除语句。其语法如下：

```
<存储模块删除语句> ::=
      <存储过程删除语句> |
```

<存储函数删除语句>

1．存储过程删除语句

从数据库中删除一个存储过程。

语法格式

DROP PROCEDURE <存储过程名定义>;
<存储过程名定义> ::=[[<数据库名>.] <模式名>.]<存储过程名>

参数

（1）<数据库名> 指明被删除的存储过程所属的数据库，缺省为当前数据库。
（2）<模式名> 指明被删除的存储过程所属的模式，缺省为当前模式。
（3）<存储过程名> 指明被删除的存储过程的名字。

使用说明

（1）如果被删除的存储过程不属于当前模式，必须在语句中指明过程的模式名。
（2)如果被删除的存储过程不属于当前数据库,还必须在语句中指明存储过程的数据库名及模式名。

权限

执行该操作的用户必须是存储过程的创建者，或者具有 DBA 系统权限。

2．存储函数删除语句

从数据库中删除一个存储函数。

语法格式

DROP FUNCTION <存储函数名定义>;
<存储函数名定义> ::=[[<数据库名>.] <模式名>.]<存储函数名>

参数

（1）<数据库名> 指明被删除的存储函数所属的数据库，缺省为当前数据库。
（2）<模式名> 指明被删除的存储函数所属的模式，缺省为当前模式。
（3）<存储函数名> 指明被删除的存储函数的名字。

使用说明

（1）如果被删除的存储函数不属于当前模式，必须在语句中指明函数的模式名。
（2)如果被删除的存储函数不属于当前数据库,还必须在语句中指明存储函数的数据库名及模式名。
（3）当模式名缺省时，默认为删除当前模式下的存储模块，否则，应指明存储模块所属的模式。除了 DBA 用户外，其他用户只能删除自己创建的存储模块。

举例说明

例 1 删除当前数据库当前模式下的存储过程 P1。

```
DROP PROCEDURE P1;
```

例 2 删除数据库 CW 的模式 AAA 下的存储函数 F1，执行该语句的用户应为函数的创建者。

```
DROP FUNCTION AAA.F1;
```

11.3 存储模块的控制语句

DMPL/SQL 语言对三种控制结构（分支结构、迭代结构和顺序结构）都提供了相应的语句支持，其语法参照了 ORACLE 7.3.4 版。控制语句具体包括：

（1）语句块
（2）赋值语句
（3）IF 语句

（4）LOOP 语句
（5）WHILE 语句
（6）FOR 语句
（7）EXIT 语句
（8）调用语句
（9）RETURN 语句
（10）NULL 语句
（11）GOTO 语句
（12）RAISE 语句
（13）打印语句
以下详细介绍各种控制语句的语法及使用方法。

11.3.1　语句块

语句块是 DMPL/SQL 语言的基本程序单元。每个语句块由关键字 DECLARE、BEGIN、EXCEPTION 和 END 划分为说明部分、执行部分和异常处理部分。其中执行部分是必须的，说明部分和异常处理部分可以省略。用户可能已经发现，事实上整个存储模块就是一个语句块，只是其说明部分省略了 DECLARE 关键字而已。语句块可以嵌套，它可以出现在任何其他语句可以出现的位置。其语法如下：

```
[DECLARE
  变量、游标或子程序等的申明]
BEGIN
  执行代码
[EXCEPTION
  异常处理代码]
END
```

在语句块的说明部分可以定义变量、游标、异常变量、子过程或子函数。执行部分是语句块的执行代码。异常处理部分处理语句块中出现的异常。这些部分的语法规则与存储模块是一致的，这里不再赘述。需要强调的一点是，一个语句块意味着一个作用域范围。也就是说，在一个语句块的说明部分定义的任何对象，其作用域就是该语句块。请看下面的例子，该例中有一个全局变量 x，同时子语句块中又定义了一个局部变量 x：

```
CREATE OR REPLACE PROCEDURE PROC_BLOCK AS
    x INT := 0;
    counter INT := 0;
  BEGIN
    FOR i IN 1 .. 4 LOOP
        x := x + 1000;
        counter := counter + 1;
        PRINT CAST(x AS CHAR(10)) || CAST(counter AS CHAR(10)) || 'OUTER loop';
        /*  这里是一个嵌套的子语句块的开始  */
      DECLARE
          x INT := 0;    --  局部变量 x，与全局变量 x 同名
      BEGIN
          FOR i IN 1 .. 4 LOOP
              x := x + 1;
              counter := counter + 1;
```

```
                PRINT CAST(x AS CHAR(10)) || CAST(counter AS CHAR(10)) || 'inner loop';
            END LOOP;
        END;
        /*  子语句块结束  */
        END LOOP;
    END;
```

执行此过程，其打印信息为：

```
1000    1       OUTER loop
1       2       inner loop
2       3       inner loop
3       4       inner loop
4       5       inner loop
2000    6       OUTER loop
1       7       inner loop
2       8       inner loop
3       9       inner loop
4       10      inner loop
3000    11      OUTER loop
1       12      inner loop
2       13      inner loop
3       14      inner loop
4       15      inner loop
4000    16      OUTER loop
1       17      inner loop
2       18      inner loop
3       19      inner loop
4       20      inner loop
```

由执行结果可以看出，两个 x 的作用域是完全不同的。

11.3.2　赋值语句

赋值语句的语法如下：

```
<赋值对象> := <表达式>  或
SET <赋值对象> = <表达式>
```

使用赋值语句可以给各种数据类型的对象赋值。被赋值的对象可以是变量，也可以是 OUT 参数或 IN OUT 参数。表达式的数据类型必须与赋值对象的数据类型兼容。

请看下面的例子：

```
CREATE OR REPLACE PROCEDURE PROC_ASSIGN AS
        V1      INT;
        V2      FLOAT;
        V3      NUMERIC;
        V4      VARCHAR(20);
        V5      DATE;
        V6      TIME;
        V7      TIMESTAMP;
```

```
        V8        INTERVAL YEAR TO MONTH;
    BEGIN
        V1:=15;
        V2:=2.3456;
        V3:=V1+V2;
        V4:= '0123456789' || CAST(V1 AS VARCHAR(10));
        V5:= DATE '1998-06-12';
        V6:=TIME '08:09:20.12345678';
        V7:=TIMESTAMP '2000-08-14 11:12:23.45678901+02:21';
        V8:= INTERVAL '0022-11' YEAR TO MONTH;
    END;
```

11.3.3　条件语句

存储模块中的 IF 语句控制执行基于布尔条件的语句序列，以实现条件分支控制结构。其语法如下：

```
IF <条件表达式>
THEN < SQL 过程语句序列>
[{ELSEIF|ELSIF <条件表达式> THEN < SQL 过程语句序列>} …]
[ELSE < SQL 过程语句序列>]
[END IF]
END IF
```

考虑到不同用户的编程习惯，ELSEIF 子句的起始关键字既可写作 ELSEIF，也可写作 ELSIF。

条件表达式中的因子可以是布尔类型的参数、变量，也可以是条件谓词。存储模块的控制语句中支持的条件谓词有：比较谓词、BETWEEN、IN、LIKE 和 IS NULL。以下就以 IF 语句为例，列举条件表达式里谓词的用法，更详细的谓词用法请参阅本手册的相关章节。

例 含 BETWEEN 谓词的条件表达式：

```
CREATE OR REPLACE PROCEDURE P_CONDITION1 (A INT) AS
BEGIN
   IF A BETWEEN -5 AND 5 THEN
     PRINT 'TRUE';
   ELSE
     PRINT 'FALSE';
   END IF;
END;
```

例 含 IN 谓词的条件表达式：

```
CREATE OR REPLACE PROCEDURE P_CONDITION2 (A INT) AS
BEGIN
   IF A IN (1,3,5,7,9) THEN
     PRINT 'TRUE';
   ELSE
     PRINT 'FALSE';
   END IF;
END;
```

例 含 LIKE 谓词的条件表达式：

```
CREATE OR REPLACE PROCEDURE P_CONDITION3(A VARCHAR(10)) AS
```

```
BEGIN
   IF A LIKE '%DM%' THEN
     PRINT 'TRUE';
   ELSE
     PRINT 'FALSE';
   END IF;
END;
```

例 含 IS NULL 谓词的条件表达式：

```
CREATE OR REPLACE PROCEDURE P_CONDITION4 (A INT) AS
BEGIN
   IF A IS NOT NULL THEN
     PRINT 'TRUE';
   ELSE
     PRINT 'FALSE';
   END IF;
END;
```

11.3.4　循环语句

存储模块支持三种基本类型的循环语句，即 LOOP 语句、WHILE 语句和 FOR 语句。LOOP 语句循环重复执行一系列语句，直到 EXIT 语句终止循环为止；WHILE 语句循环检测一个条件表达式，当表达式的值为 TRUE 时就执行循环体的语句；FOR 语句对一系列的语句重复执行指定次数的循环。以下对这三种语句分别进行介绍。

1．LOOP 语句

LOOP 语句的语法如下：

```
LOOP
    语句序列
END LOOP
```

LOOP 语句实现对一语句系列的重复执行，是循环语句的最简单形式。它没有明显的终点，必须借助 EXIT 语句来跳出循环。以下是 LOOP 语句的用法举例：

```
CREATE OR REPLACE PROCEDURE P_LOOP(A IN OUT INT) AS
     BEGIN
        LOOP
           IF A<=0 THEN
              EXIT;
           END IF;
           PRINT A;
           A:=A-1;
     END LOOP;
END;
```

第 3 行到第 9 行是一个 LOOP 循环，每一次循环都打印参数 A 的值，并将 A 的值减 1，直到 A 小于等于 0。

2．WHILE 语句

WHILE 语句的语法如下：

```
WHILE <条件表达式> LOOP
    语句序列
END LOOP
```

WHILE 循环语句在每次循环开始以前，先计算条件表达式，若该条件为 TRUE，语句序列被执行一次，然后控制重新回到循环顶部。若条件表达式的值为 FALSE，则结束循环。当然，也可以通过 EXIT 语句来终止循环。以下是 WHILE 语句的用法举例：

```
CREATE OR REPLACE PROCEDURE P_WHILE(A IN OUT INT) AS
  BEGIN
    WHILE A>0 LOOP
      PRINT A;
      A:=A-1;
     END LOOP;
  END;
```

这个例子的功能与上例相同，只是使用了 WHILE 循环结构。

3．FOR 语句

FOR 语句的语法如下：

```
FOR <循环计数器> IN [REVERSE] <下限表达式>  ..  <上限表达式> LOOP
    语句序列
END LOOP
```

循环计数器是一个标识符，它类似于一个变量，但是不能被赋值，且作用域限于 FOR 语句内部。下限表达式和上限表达式用来确定循环的范围，它们的类型必须和整型兼容。循环范围是在循环开始之前确定的，即使在循环过程中下限表达式或上限表达式的值发生了改变，也不会引起循环范围的变化。

执行 FOR 语句时，首先检查下限表达式的值是否小于上限表达式的值，如果下限数值大于上限数值，则不执行循环体。否则，将下限数值赋给循环计数器（语句中使用了 REVERSE 关键字时，则把上限数值赋给循环计数器）；然后执行循环体内的语句序列；执行完后，循环计数器值加 1（如果有 REVERSE 关键字，则减 1）；检查循环计数器的值，若仍在循环范围内，则重新继续执行循环体；如此循环，直到循环计数器的值超出循环范围。同样，也可以通过 EXIT 语句来终止循环。以下是 FOR 语句的用法举例：

```
CREATE OR REPLACE PROCEDURE P_FOR1 (A IN OUT INT) AS
  BEGIN
    FOR I IN REVERSE 1 .. A LOOP
      PRINT I;
      A:=I-1;
    END LOOP;
  END;
```

这个例子的功能也与上例相同，只是使用了 FOR 循环结构。

FOR 语句中的循环计数器可与当前语句块内的参数或变量同名，这时该同名的参数或变量在 FOR 语句的范围内将被屏蔽。例子如下：

```
CREATE OR REPLACE PROCEDURE P_FOR2 AS
    V1 DATE:=DATE '2000-01-01';
  BEGIN
    FOR V1 IN 0 .. 5 LOOP
      PRINT V1;
```

```
        END LOOP;
        PRINT V1;
    END;
```

调用此过程：

```
 CALL   P_FOR2;
```

打印结果为：

0
1
2
3
4
5
DATE '2000-01-01'

此例中，循环计数器 V1 与变量 V1 同名。由调用结果可见，在 FOR 语句内，PRINT 语句将 V1 当做循环计数器。而 FOR 语句外的 PRINT 语句则将 V1 当作 DATE 类型的变量。请注意：在符号‘..’和上/下限表达式间应用空格隔开，否则容易引起编译器的误解。

11.3.5　EXIT 语句

EXIT 语句与循环语句一起使用，用于终止其所在循环语句的执行，将控制转移到该循环语句外的下一个语句继续执行。其语法如下：

```
<EXIT 语句> ::= EXIT [<标号名>] [WHEN <条件表达式>]
```

EXIT 语句必须出现在一个循环语句中，否则编译器将报错。

当 EXIT 后面的标号名省略时，该语句将终止直接包含它的那条循环语句；当 EXIT 后面带有标号名时，该语句用于终止标号名所标识的那条循环语句。需要注意的是，该标号名所标识的语句必须是循环语句，并且 EXIT 语句必须出现在此循环语句中。当 EXIT 语句位于多重循环中时，可以用该功能来终止其中的任何一重循环。

当 WHEN 子句省略时，EXIT 语句无条件的终止该循环语句；否则，先计算 WHEN 子句中的条件表达式，当条件表达式的值为真时，终止该循环语句。

前面我们已经见到了 EXIT 语句的用法，该例子还可以这样实现：

```
CREATE OR REPLACE PROCEDURE P_LOOP(A IN OUT INT) AS
  BEGIN
      LOOP
          EXIT WHEN A<=0;
          PRINT A;
          A:=A-1;
      END LOOP;
  END;
```

11.3.6　调用语句

存储模块可以被其他存储模块或应用程序调用。同样，在存储模块中也可以调用其他存储模块。调用存储过程时，应给存储过程提供输入参数值，并获取存储过程的输出参数值。调用的形式为：

语法格式

```
CALL [[<数据库名>.]<模式名>.]<存储过程名> [(<参数>{, <参数>})];
```

参数

1．<数据库名>　　指明被调用存储过程所属的数据库。
2．<模式名>　　　指明被调用存储过程所属的模式。
3．<存储过程名>　指明被调用存储过程的名字。
4．<参数>　　　　指明提供给存储过程的参数。

使用说明

1．如果被调用的存储过程不属于当前模式，必须在语句中指明存储过程的模式名。
2．如果被调用的存储过程不属于当前数据库，还必须在语句中指明存储过程的数据库名及模式名。
3．参数的个数和类型必须与被调用的存储过程一致。
4．存储过程的输入参数可以是嵌入式变量，也可以是值表达式；存储过程的输出参数必须是可赋值对象，如嵌入式变量。
5．在调用过程中，服务器将以存储过程创建者的模式和权限来执行过程中的语句。

权限

执行该操作的用户必须拥有该存储过程的 EXECUTE 权限。存储过程的所有者和 DBA 用户隐式具有该过程的 EXECUTE 权限，该权限也可通过授权语句显式授予其他用户。

如果是过程调用，调用语句必须是一条单独的控制语句；如果是函数调用，则必须出现在一个表达式当中。请注意如果是在交互式环境中调用存储过程，不能省略关键字 CALL。

例　存储过程的调用

```
--在数据库 CW 中以用户 SYSDBA 的身份创建存储过程 P1：
CREATE OR REPLACE PROCEDURE P1(A IN OUT INT) AS
 V1 INT:=A;
BEGIN
 A:=0;
 FOR B IN 1 .. V1 LOOP
 A:=A+B;
   END LOOP;
END;
在存储过程 P2 中调用存储过程 P1：
CREATE OR REPLACE PROCEDURE P2(A IN INT) AS
 V1 INT :=A;
BEGIN
 CW.SYSDBA.P1(V1);
 PRINT V1;
END;
```

例　存储函数的调用

```
创建存储函数 MAXNUM：
CREATE OR REPLACE FUNCTION MAXNUM(A INT, B INT) AT CW RETURN INT AS
BEGIN
 IF (A >= B) THEN
   RETURN A;
 ELSE
   RETURN B;
 END IF;
```

```
END;

假设存在表 T_NUM(COL INT)，在存储过程 PROC_MAXNUM 中调用存储函数 MAXNUM：
CREATE OR REPLACE PROCEDURE PROC_MAXNUM(A INT, B INT) AS
BEGIN
  INSERT INTO T_NUM VALUES(MAXNUM(A,B));
  PRINT MAXNUM(A,B);
END;
```

11.3.7　RETURN 语句

RETURN 语句的语法如下：

```
RETURN [<返回值>]
```

其功能为：结束模块的执行，将控制返回给该模块调用者。如果是从函数返回，则同时将函数的返回值提供给调用环境。

需要强调的是，函数的执行必须以 RETURN 语句结束。确切的说，函数中应至少有一个被执行的返回语句。由于程序中有分支结构，在编译时很难保证其每条路径都有返回语句，因为我们在函数执行时才对其进行检查。

例子如下：

```
CREATE OR REPLACE FUNCTION FUN1(A INT) RETURN VARCHAR(10) AS
BEGIN
    IF (A<0) THEN
      RETURN ' A<0';
    ELSE
      NULL;
    END IF;
END;
```

此例中，当参数 A 大于或等于 0 时，函数 FUN1 没有以 RETURN 语句结束。因此，执行下面的语句：

```
SELECT FUN1(1);
```

服务器将报错：函数没有返回参数。

11.3.8　NULL 语句

NULL 语句的语法为：

```
NULL
```

该语句不做任何事情，只是用于保证语法的正确性，或增加程序的可读性。

11.3.9　GOTO 语句

GOTO 语句无条件地跳转到一个标号所在的位置。标号的定义在一个语句块中必须是唯一的。其语法如下：

```
GOTO <标号名>
```

GOTO 语句将控制权交给带有标号的语句或语句块。为了保证 GOTO 语句的使用不会引起程序的混

乱，我们对 GOTO 语句的使用有下列限制：

GOTO 语句不能跳入一个 IF 语句、循环语句或下层语句块中；

GOTO 语句不能从一个异常处理器跳回当前块，但是可以跳转到包含当前块的上层语句块。

下面是一些非法的 GOTO 语句的例子。

例
```
BEGIN
    ...
    GOTO update_row;      /*  错误，企图跳入一个 IF 语句  */
    ...
    IF valid THEN
        ...
        <<update_row>>
        UPDATE emp SET ...
    END IF;
END;
```

例
```
BEGIN
    ...
    IF valid THEN
        ...
        GOTO update_row;   /*  错误，企图从 IF 语句的一个子句跳入另一个子句  */
    ELSE
        ...
        <<update_row>>
        UPDATE emp SET ...
    END IF;
END;
```

例
```
BEGIN
    ...
    IF status = 'OBSOLETE' THEN
        GOTO delete_part;    /*  错误，企图跳入一个下层语句块  */
    END IF;
    ...
    BEGIN
        ...
        <<delete_part>>
        DELETE FROM parts WHERE ...
    END;
END;
```

11.3.10　RAISE 语句

语法：

```
RAISE <异常名>
```

RAISE 语句用于抛出异常。要查看详细的说明，请阅读第 11.4 节“存储模块的异常处理。”

11.3.11　打印语句

打印语句用于从存储模块中向客户端输出一个字符串，方便用户调试存储模块代码。当存储模块的行为与预期的不一致时，可以在其中放进打印语句来观察各个阶段的运行情况。打印语句的语法为：

```
PRINT <表达式>
```

打印语句中的表达式可以是各种数据类型，系统自动将其转换为字符类型。前面的例子中有多处用到了打印语句，用户可自行参考，这里不再另外举例。

11.4　存储模块的异常处理

在存储模块的正常执行过程中，可能会出现未预料的事件，这就是异常（EXCEPTION）。异常事件会导致代码的不正确结束，因此用户需要在程序中对异常进行处理，以保证不可预知的错误不会终止PL/SQL 模块。为此，DM 提供了异常处理机制，使用户可以在语句块的异常处理部分中处理异常。这里所说的异常，可以是系统预定义异常，也可以是用户自定义异常。

为方便用户编程，DM 系统提供了一些预定义的异常，这些异常与最常见的 DM 错误相对应。下表中列举了 DM 系统的预定义异常：

异 常 名 称	SQLCODE	说　　明
DUP_VAL_ON_INDEX	-2310	违反唯一性约束
INVALID_CURSOR	-3029	对游标的操作非法
TOO_MANY_ROWS	-1091	SELECT INTO 语句返回的记录数大于 1
ZERO_DIVIDE	-2205	除零错误

此外，还有一个特殊的异常名 OTHERS，它处理所有没有明确列出的异常。OTHERS 对应的异常处理语句必须放在其它异常处理语句之后。

11.4.1　异常变量的说明

如果用户需要处理非预定义的异常，可以自己定义一个异常。创建自定义异常的方法是：在程序块的说明部分定义一个异常变量，并将该异常变量与用户要处理的 DM 错误号绑定。异常变量说明的语法如下：

```
<异常变量名> EXCEPTION [FOR <错误号>]
```

其中，FOR 子句用来为异常变量绑定错误号（SQLCODE 值）。

异常变量类似于一般的变量，必须在块的说明部分说明，有同样的生存期和作用域。但是异常变量不能作参数传递，也不能被赋值。

需要注意的是，为异常变量绑定的错误号不一定是 DM 返回的系统错误，但是该错误号必须是一个负整数。自定义异常使得用户可以把违背事务规则的行为也作为异常来看待。

11.4.2　异常的抛出

在存储模块的执行中如果发生错误，系统将自动抛出一个异常。此外，我们还可以用 RAISE 语句自己抛出异常，例如，当操作合法，但是违背了事务规则时。一旦异常被抛出，执行就被传递给程序块的异常处理部分。RAISE 语句的语法如下：

```
RAISE <异常名>
```

其中，<异常名>可以是系统预定义异常，也可以是用户自定义异常。

11.4.3　异常处理器

程序块的异常处理部分可以处理一个或者多个异常。其语法如下：

```
<异常处理部分> ::= EXCEPTION    {<异常处理语句>;}…

<异常处理语句> ::= WHEN <异常名> THEN <语句序列>
```

EXCEPTION 关键字表示异常处理部分的开始，异常处理器即异常处理语句中 WHEN 子句后面的部分。如果在语句块的执行中出现异常，执行就被传递给语句块的异常处理部分。而如果在本语句块的异常处理部分没有相应的异常处理器对它进行处理，系统就会中止此语句块的执行，并将此异常传递到该语句块的上一层语句块或其调用者，这样一直到最外层。如果始终没有找到相应的异常处理器，则中止本次调用语句的处理，并向用户报告错误。

异常处理部分是可选的。但是如果出现 EXCEPTION 关键字时，必须至少有一个异常处理器。异常处理器可以按任意次序排列，只有 OTHERS 异常处理器必须在最后，它处理所有没有明确列出的异常。此外，同一个语句块内的异常处理器不允许处理重复的异常。

11.4.4　异常处理用法举例

下面的过程中，由于出现了除零错误，系统将抛出异常 ZERO_DIVIDE。在异常处理部分，我们定义了一个异常处理器来处理该异常。例子如下：

```
CREATE OR REPLACE PROCEDURE SYSEXCEPT1 AS
      A INTEGER:=1;
 BEGIN
      A:=A/0;  /*  除零错误  */
 EXCEPTION
      WHEN ZERO_DIVIDE THEN
         PRINT 'Divided by zero';
 END;
```

执行此过程，服务器返回打印信息：‘Divided by zero’。

下层语句块中出现的异常，如果在该语句块中没有对应的异常处理器，可以被上层语句块或其调用者的异常处理器处理。如下例中，一个下层语句块中抛出了异常 e1，但是没有处理该异常。那么该异常将由它的上层语句块中的异常处理器来处理：

```
CREATE OR REPLACE PROCEDURE EXCEPT1 AS
   E1 EXCEPTION;
 BEGIN
   PRINT 'Before sub block';
   /*  下层语句块开始  */
   BEGIN
      RAISE E1;                  /*  抛出异常 E1 */
   END;
   /*  下层语句块结束  */
   PRINT 'After sub block ';
 EXCEPTION
```

```
    WHEN E1 THEN
      PRINT 'Exception E1 raised from sub block catched';
END;
```

执行此过程，服务器返回打印信息：

Before sub block

exception E1 raised from sub block catched

要注意异常变量的有效范围。在上例中，异常变量 E1 的有效范围是整个存储过程 EXCEPT1。如果改为：

```
CREATE OR REPLACE PROCEDURE EXCEPT2 AS
BEGIN
  PRINT 'Before sub block ';
  /*  下层语句块开始  */
  DECLARE
    E1 EXCEPTION;          /*  异常变量定义在下层语句块中  */
  BEGIN
    RAISE E1;
  END;
  /*  下层语句块结束  */
  PRINT 'After sub block ';
EXCEPTION
  WHEN E1 THEN             /*  错误：异常变量 E1 在此范围内没有定义  */
    PRINT 'Exception E1 raised from sub block catched';
END;
```

创建该过程时，编译器将报错：无效的异常名。

有时候，在异常处理器的执行中又可能出现新的异常。这时，系统将新出现的异常作为当前需要处理的异常，并向上层传递。如下例：

```
CREATE OR REPLACE PROCEDURE EXCEPT3 AS
  E1 EXCEPTION FOR -10000;
  E2 EXCEPTION FOR -10001;
BEGIN
  BEGIN
    RAISE E1;          /*  下层语句块抛出异常 E1 */
  EXCEPTION
    WHEN E1 THEN
      RAISE E2; /*  在异常 E1 的处理过程中，抛出异常 E2 */
  END
EXCEPTION
  WHEN E1 THEN      /*  异常变量 E1 的异常处理器  */
    PRINT 'Exception E1 catched';
  WHEN E2 THEN      /*  异常变量 E2 的异常处理器  */
    PRINT 'Exception E2 catched';
END;
```

执行此过程，服务器将返回打印信息：

'Exception E2 catched'

因为下层语句块在处理异常 E1 的过程中，又抛出了异常 E2。那么，E2 将作为当前需要处理的异常被传递到其上层语句块中。因此，在上层语句块中，E2（而不是 E1）的异常处理器将被执行。

11.5 存储模块的 SQL 语句

存储模块中支持的 SQL 语句包括：

（1）数据定义语句（支持临时表的创建与删除）

（2）数据查询语句（SELECT）

（3）数据操纵语句（INSERT、DELETE、UPDATE）

（4）游标定义及操纵语句（DECLARE CURSOR、OPEN、FETCH、CLOSE）

（5）事务控制语句（COMMIT、ROLLBACK）

（6）动态 SQL 执行语句（EXECUTE IMMEDIATE）

SQL 语句中可以包含变量、参数和存储函数。DM 系统允许变量或参数与 SQL 语句中的表、视图及列同名，但是为了避免混淆及降低程序的可读性，建议用户尽量避免这种用法。

11.5.1 游标

存储模块中可以定义和操纵游标。

游标定义必须出现在块的定义部分。它与变量处于同一个的命名空间，有着相同的作用域。但是游标不能作参数传递，也不能被赋值。

每个游标都有 4 个属性，分别是 FOUND、NOTFOUND、ISOPEN 和 ROWCOUNT，用于描述其状态。具体说明见下表：

属性名	说　　明
FOUND	如果游标未打开，产生一个异常。否则，在第一次拨动游标之前，其值为 NULL。如果最后一次拨动游标时取到了数据，其值为真，否则为假。
NOTFOUND	如果游标未打开，产生一个异常。否则，在第一次拨动游标之前，其值为 NULL。如果最后一次拨动游标时取到了数据，其值为假，否则为真。
ISOPEN	游标打开时为真，否则为假。
ROWCOUNT	如果游标未打开，产生一个异常。如游标已打开，在第一次拨动游标之前，其值为 0，否则为最后一次拨动后已经取到的元组数。

以上的游标属性称为显式游标属性，其用法如下：

```
<游标名>%<游标属性>
```

下面是显式游标属性的用法举例。假设有表 EMP，表中的数据如下：

ENAME	EMPNO	SAL
KING	7839	5000
SCOTT	7788	3000
FORD	7902	3000
JONES	7566	2975
BLAKE	7698	2850
CLARK	7782	2450
ALLEN	7499	1600
TURNER	7844	1500

MILLER	7934	1300
WARD	7521	1250
MARTIN	7654	1250
ADAMS	7876	1100
JAMES	7900	950
SMITH	7369	800

例　对于基表 EMP(ename char(10), empno numeric(4), sal numeric(4))，输出表中的前 5 行数据。如果表中的数据不足 5 行，则输出表中的全部数据。

```
CREATE OR REPLACE PROCEDURE PROC_CURSOR AS
   c1 CURSOR FOR SELECT * FROM EMP;
   my_ename    CHAR(10);
   my_empno    NUMERIC(4);
   my_sal          NUMERIC(7,2);
BEGIN
  OPEN c1;
  LOOP
     FETCH c1 INTO my_ename, my_empno, my_sal;
     EXIT WHEN c1%NOTFOUND;           /*  如表中数据不足 5 行，跳出循环  */
     PRINT my_ename || '    ' || my_empno || '    ' || my_sal;
     EXIT WHEN c1%ROWCOUNT=5;      /*  已经输出了 5 行数据，跳出循环*/
  END LOOP;
  CLOSE c1;
END;
```

除了显式游标属性外，DM 中还有一个隐式的游标 SQL，该游标保存了最近一次 INSERT、UPDATE、DELETE 或 SELECT INTO 操作的信息。它的属性称为隐式游标属性，具体说明见下表：

属性名	说　　明
FOUND	如果最近的数据操纵语句影响了数据库中的元组，或者查询语句返回了一行以上的结果，其值为真，否则为假。
NOTFOUND	如果最近的数据操纵语句影响了数据库中的元组，或者查询语句返回了一行以上的结果，其值为假，否则为真。
ISOPEN	其值始终为假。
ROWCOUNT	如果最近的数据操纵语句影响了数据库中的元组，或者查询语句返回了一行以上的结果，其值为数据操纵语句影响的元组数或查询语句返回的元组数。否则，其值为 0。

如果系统还未执行过任何数据操纵或查询操作，则隐式游标属性的值均为 NULL。隐式游标属性的用法如下：

```
SQL%<游标属性>
```

下面是隐式游标属性的一些用法举例：

例　如果 UPDATE 操作没有影响任何元组，则插入一条元组。

```
UPDATE emp SET sal = sal * 1.05 WHERE empno = my_empno;
IF SQL%NOTFOUND THEN
     INSERT INTO emp VALUES (my_empno, my_ename,my_sal);
END IF;
```

例　如果 DELETE 操作影响的元组数超过 100，则抛出一个异常。

```
DELETE FROM parts WHERE status = ' OBSOLETE';
IF SQL%ROWCOUNT > 100 THEN    -- more than 100 rows were deleted
    RAISE large_deletion;
END IF;
```

11.5.2　动态 SQL

动态 SQL 功能在许多应用中得到广泛使用。存储模块中也提供了动态执行 SQL 语句的功能。使用 EXECUTE IMMEDIATE 语句，可准备并立即执行一条动态 SQL 语句。其语法如下：

```
EXECUTE IMMEDIATE <SQL 动态语句文本>
[INTO <赋值对象> {,<赋值对象>}]
[USING <绑定参数> {,<绑定参数>}]
```

INTO 子句仅用于动态 SQL 语句为单行查询的情况。其中，<赋值对象>用来存储查询语句返回的数据。<赋值对象>可以是变量，也可以是 OUT 或 IN OUT 类型的参数，其个数与类型应与查询返回结果的各列相对应。

<SQL 动态语句文本>中可以有参数，每个形参的形式为一个‘?’符号。通过 USING 子句可以指定每个参数对应的实参值。 <绑定参数>即每个参数对应的实参，它可以是存储模块的参数或者变量，其个数与出现顺序应与形参对应。

用户可以通过绑定不同的实参来重复执行一条动态 SQL 语句。但是要知道每次重复执行时，系统都会重新准备该语句后再执行，所以这样做并不能降低系统开销。

以下是动态 SQL 的例子：

```
   CREATE OR REPLACE PROCEDURE P_DYN_SQL AS
     sql_stmt   VARCHAR(100);
     my_ename       VARCHAR(15) := 'Jerry';
     my_empno       NUMERIC(4):= '7963';
     my_sal     MONEY:=2450;
 BEGIN
     sql_stmt := 'INSERT INTO emp VALUES (?, ?, ?)';
     EXECUTE IMMEDIATE sql_stmt USING my_ename, my_empno, my_sal;

     sql_stmt := 'SELECT sal FROM emp WHERE empno = ?';
     my_empno := 7788;
     EXECUTE IMMEDIATE sql_stmt INTO my_sal USING my_empno;

     my_empno := 7369;
     EXECUTE IMMEDIATE 'DELETE FROM emp WHERE empno = ?' USING my_empno;
 END;
```

11.5.3　游标变量

与普通游标不同，游标变量可以指向不同的游标工作区，它为用户提供了更灵活的数据操作方法。定义和打开游标变量的语法与普通游标不同。

定义游标变量：

```
<游标变量名> CURSOR
```

打开游标变量：

```
OPEN <游标变量名> FOR <查询语句>
```

或

```
OPEN <游标变量名> FOR <查询语句文本>
```

其它的游标操作与普通游标相同，绑定参数的内容见前述动态 SQL 章节。
例子如下：

```
CREATE OR REPLACE PROCEDURE curvar AS
    c1        CURSOR;
    vename      CHAR(10);
    vempno      NUMERIC(4);
    vsal        NUMERIC(7,2);
    BEGIN
      vsal:=2000;
      OPEN c1 FOR 'SELECT ENAME, EMPNO FROM EMP where SAL> vsal';
      LOOP
          FETCH c1 INTO vename, vempno;
              EXIT WHEN c1%NOTFOUND;
                  PRINT 'NAME = '|| vename ||'    NO = '||vempno;
         END LOOP;
      CLOSE c1;
    END;
```

本例中的OPEN语句也可写成如下等价的形式：

```
OPEN c1 FOR SELECT ENAME, EMPNO FROM EMP where SAL > vsal;
```

11.5.4　返回查询结果集

在存储过程中如果执行了不带INTO子句的查询语句，系统将在调用结束时将该查询结果集返回给调用者。当出现多个查询语句时，只有最后被执行的查询语句的查询结果集被返回。下面给出一个例子：

```
CREATE OR REPLACE PROCEDURE p_sel_result(UserTag byte) AS
    BEGIN
      IF (UserTag = 1) THEN
              SELECT * FROM SYSTABLES WHERE ID > 1000;
      ELSE
          SELECT * FROM SYSTABLES WHERE ID < 1000;
      END IF;
    END;
```

11.5.5　SQL 语句应用举例

下面我们给出一个SQL语句应用的例子。

本例中，有两个表ACCOUNT和ACTIONS。我们将根据表ACTIONS中的指令来修改表ACCOUNT中的数据。表ACTIONS中的数据包括：要修改的表ACCOUNT中元组的ID、要做的操作（I表示INSERT，D表示DELETE，U表示UPDATE）和修改后的新值。此外，还有一列用于记录操作是否成功的状态信息。

对于INSERT操作，如果表ACCOUNT中已存在指定的元组，则直接更新此元组的值；对于UPDATE

操作，如果表ACCOUNT中不存在指定的元组，则插入一条新的元组；对于DELETE操作，如果表ACCOUNT中不存在指定的元组，则不做任何操作。

表ACCOUNT中的初始数据如下，其中ACCOUNT_ID列为主关键字：

ACCOUNT_ID	BAL
1	1000
2	2000
3	1500
4	6500
5	500

表ACTIONS中的初始数据如下：

ACCOUNT_ID	OPER_TYPE	NEW_VALUE	STATUS
3	U	599	
6	I	20099	
5	D		
7	U	1599	
1	I	399	
9	D		
10	X		

例

```
CREATE OR REPLACE PROCEDURE PROC_MODIFY AS
    c1 CURSOR FOR SELECT account_id, oper_type, new_value FROM actions;
    v_account_id      INT;
    v_oper_type       CHAR;
    v_new_value       NUMERIC(10,2);
 BEGIN
   OPEN c1;
   LOOP    --  执行表 ACTIONS 中的每个操作
     FETCH c1 INTO v_account_id, v_oper_type, v_new_value;
     EXIT WHEN c1%NOTFOUND;
     v_oper_type := UPPER(v_oper_type);

     IF v_oper_type = 'U' THEN
       UPDATE account SET bal = v_new_value WHERE account_id = v_account_id;

       IF SQL%NOTFOUND THEN    --  如果元组不存在，插入一条新的元组
           INSERT INTO account
                VALUES (v_account_id, v_new_value);
           UPDATE actions SET status = 'Update: ID not found. Value inserted. '
                WHERE CURRENT OF c1;
       ELSE
           UPDATE actions SET status = 'Update: Success. '
                WHERE CURRENT OF c1;
       END IF;

 ELSIF v_oper_type = 'I' THEN
```

```
            BEGIN
                INSERT INTO account
                VALUES (v_account_id, v_new_value);
               UPDATE actions set status = 'Insert: Success. '
                     WHERE CURRENT OF c1;
           EXCEPTION
               WHEN DUP_VAL_ON_INDEX THEN   --  如元组已存在，更新此元组
                  UPDATE account SET bal = v_new_value
                        WHERE account_id = v_account_id;
                  UPDATE actions SET status = 'Insert: Acct exists. Updated instead'
                        WHERE CURRENT OF c1;
           END;

         ELSIF v_oper_type = 'D' THEN
            DELETE FROM account
                  WHERE account_id = v_account_id;

            IF SQL%NOTFOUND THEN      --  元组不存在
               UPDATE actions SET status = 'Delete: ID not found. '
                     WHERE CURRENT OF c1;
           ELSE
               UPDATE actions SET status = 'Delete: Success. '
                     WHERE CURRENT OF c1;
           END IF;

        ELSE    --  无效的操作
           UPDATE actions SET status = 'Invalid operation. No action taken. '
                 WHERE CURRENT OF c1;
        END IF;
     END LOOP;
     CLOSE c1;
     COMMIT;
  END;
```

执行此过程：

```
CALL PROC_MODIFY;
```

现在表 ACCOUNT 中的数据应为：

ACCOUNT_ID	BAL
1	399
2	2000
3	599
4	6500
6	20099
7	1599

表 ACTIONS 中的数据应为：

ACCOUNT_ID	OPER_TYPE	NEW_VALUE	STATUS

3	U	599	Update: Success.
6	I	20099	Insert: Success.
5	D		Delete: Success.
7	U	1599	Update: ID not found. Value inserted.
1	I	399	Insert: Acct exists. Updated instead.
9	D		Delete: ID not found.
10	X		Invalid operation. No action taken.

第 12 章　触发器

DM 是一个具有主动特征的数据库管理系统，其主动特征包括约束机制和触发器机制。通过触发器机制，用户可以定义、删除和修改触发器。DM 自动管理和运行这些触发器，从而体现系统的主动性，方便用户使用。

触发器（TRIGGER）定义当某些与数据库有关的事件发生时，数据库应该采取的操作。这些事件包括某个基表上的 INSERT、DELETE 和 UPDATE 操作。触发器与存储模块类似，都是在服务器上保存并执行的一段 PL/SQL 语句。不同的是：存储模块必须被显式地调用执行，而触发器是在相关的事件发生时由服务器自动地隐式地激发。触发器是激发它们的语句的一个组成部分，即直到一个语句激发的所有触发器执行完成之后该语句才结束，而其中任何一个触发器执行的失败都将导致该语句的失败，触发器所做的任何工作都属于激发该触发器的语句。

触发器为用户提供了一种自己扩展数据库功能的方法。关于触发器应用的例子有：

（1）利用触发器实现表约束机制（如：PRIMARY KEY、FOREIGN KEY、CHECK 等）无法实现的复杂的引用完整性；

（2）利用触发器实现复杂的事务规则（如：想确保薪水增加量不超过 25%）；

（3）利用触发器维护复杂的缺省值（如：条件缺省）；

（4）利用触发器实现复杂的审计功能；

（5）利用触发器防止非法的操作。

触发器是应用程序分割技术的一个基本组成部分，它将事务规则从应用程序的代码中移到数据库中，从而可确保加强这些事务规则并提高它们的性能。

在本章各例中，**如不特别说明，各例的当前数据库均为 CW，用户均为建表者 USER1，因此表名前通常省略了数据库名、模式名前缀。**

12.1　触发器的定义

用户可使用触发器定义语句（CREATE TRIGGER）在一张基表上创建触发器。下面是触发器定义语句的语法：

语法格式

```
CREATE [OR REPLACE] TRIGGER <触发器名>
BEFORE | AFTER <触发事件> {OR <触发事件>} ON <触发表名>
[WITH    ENCRYPTION]
[REFERENCING
      OLD [ROW] [AS] <引用变量名> | NEW [ROW] [AS] <引用变量名>
      {, OLD [ROW] [AS] <引用变量名> | NEW [ROW] [AS] <引用变量名>}]
[FOR EACH {ROW | STATEMENT}]
[WHEN <条件表达式>]
<触发器体>
```

```
<触发表名>::=[[<数据库名>.] <模式名>.]<基表名>
```

参数

1．<触发器名>　　指明被创建的触发器的名称。

2．BEFORE　　　指明触发器在执行触发语句之前激发。

3．AFTER　　　指明触发器在执行触发语句之后激发。

4．<触发事件>　　指明激发触发器的事件，可以是 INSERT、DELETE 或 UPDATE，其中 UPDATE 事件可通过 UPDATE OF <触发列清单>的形式来指定所修改的列。

5．<基表名>　　　指明被创建触发器的基表的名称。

6．REFERENCING 子句　指明相关名称可以在元组级触发器的触发器体和 WHEN 子句中利用相关名称来访问当前行的新值或旧值，缺省的相关名称为 OLD 和 NEW。

7．<引用变量名>　　标识符，指明行的新值或旧值的相关名称。

8．FOR EACH 子句　指明触发器为元组级或语句级触发器。FOR EACH ROW 表示为元组级触发器，它为被触发命令影响、且 WHEN 子句的表达式计算为真的每条记录激发一次。FOR EACH STATEMENT 为语句级触发器，它对每个触发命令执行一次。语句级触发器的 FOR EACH 子句可以省略。

9．WHEN 子句　　　只允许为元组级触发器指定 WHEN 子句，它包含一个布尔表达式，当表达式的值为 TRUE 时，执行触发器；否则，跳过该触发器。

10．<触发器体>　　触发器被触发时执行的 SQL 过程语句块。

功能

创建数据库触发器，并使其处于允许状态。

使用说明

1．<触发器名>是触发器的名称，它不能与模式内的其他触发器同名。

2．可以使用 OR REPLACE 选项来替换一个触发器，但是要注意被替换的触发器的触发表不能改变。如果要在同一模式内不同的表上重新创建一个同名的触发器，则必须先删除该触发器，然后再创建。

3．<触发事件子句>说明激发触发器的事件；<触发器体>是触发器的执行代码；<引用子句>用来引用正处于修改状态下的行中的数据。如果指定了<触发条件>子句，则首先对该条件表达式求值，<触发器体>只有在该条件为真值时才运行。<触发器体>是一个 DMPL/SQL 语句块，它与存储模块定义语句中<模块体>的语法基本相同。有关详细语法，用户可参考第 11 章的相关部分。

4．在一张基表上允许创建的触发器的个数没有限制，一共可能有 12 种类型。它们分别是：BEFORE INSERT 行级、BEFORE INSERT 语句级、AFTER INSERT 行级、AFTER INSERT 语句级、BEFORE UPDATE 行级、BEFORE UPDATE 语句级、AFTER UPDATE 行级、AFTER UPDATE 语句级、BEFORE DELETE 行级、BEFORE DELETE 语句级、AFTER DELETE 行级和 AFTER DELETE 语句级。

5．触发器是在 DML 语句运行时激发的。执行 DML 语句的算法步骤如下：

（1）如果有语句级前触发器的话，先运行该触发器；

（2）对于受语句影响每一行：

①如果有行级前触发器的话，运行该触发器；

②执行该语句本身；

③对于受语句影响每一行，如果有行级后触发器的话，运行该触发器；

（3）如果有语句级后触发器的话，运行该触发器。

6．元组级触发器体中的 SQL 语句不能读或修改触发语句的变异表。所谓变异表，就是当前被 DML 语句修改的表。对触发器来说，变异表就是创建该触发器的表。但是，对于值插入语句，元组级触发器不把触发表作为变异表对待。

权限

使用该语句的用户必须是相应基表的创建者，或者具有 DBA 系统权限。

需要强调的是，由于触发器是激发它们的语句的一个组成部分，为保证语句的原子性，在<触发器体>以及<触发器体>调用的存储模块中不允许使用可能导致事务提交或回滚的 SQL 语句。具体地说，在触发器中允许的 SQL 语句有：SELECT、INSERT、DELETE、UPDATE、DECLARE CURSOR、OPEN、FETCH 和 CLOSE 语句。

每张基表上的可创建的触发器的个数没有限制，但是触发器的个数越多，处理 DML 语句所需的时间就越长，这是显而易见的。创建触发器的用户必须是基表的创建者，或者拥有 DBA 权限。注意，不存在触发器的执行权限，因为用户不能主动“调用”某个触发器，是否激发一个触发器是由系统来决定的。

12.1.1　触发器类型

一个触发器的类型由该触发器的触发事件、级别及其执行的时间共同决定。下面分别具体介绍。

1．INSERT、DELETE 和 UPDATE 触发器

激发触发器的触发事件可以是三种数据操作命令，即 INSERT、DELETE 和 UPDATE 操作。在触发器定义语句中用关键字 INSERT、DELETE 和 UPDATE 指明构成一个触发器事件的数据操作的类型，其中 UPDATE 触发器可能依赖于所修改的列，在定义中可通过 UPDATE OF <触发列清单>的形式来指定所修改的列。如下例所示：

```
先建表 T1:
CREATE TABLE T1(
COL1    CHAR(10)    NOT NULL ,
COL2    CHAR(23) ,
COL3    CHAR(40));
在 T1 上建立触发器：
CREATE OR REPLACE TRIGGER TRG_UPD
    AFTER UPDATE OF COL1,COL3 ON T1
    BEGIN
        PRINT 'UPDATE OPERATION ON COLUMNS COL1 OR COL3 OF T1';
    END;
```

当对表 T1 进行更新操作，并且更新的列中包括 COL1 或 COL3 时，此例中定义的触发器 TRG_UPD 将被激发。

如果一个触发器的触发事件为 INSERT，则该触发器被称为 INSERT 触发器，同样

也可以这样来定义 DELETE 触发器和 UPDATE 触发器。一个触发器的触发事件也可以是多个数据操作命令的组合，这时这个触发器可由多种数据操作命令激发。如下例所示：

```
CREATE OR REPLACE TRIGGER TRG_INS_DEL
 AFTER INSERT OR DELETE ON T1
 BEGIN
    PRINT 'INSERT OR DELETE OPERATION ON T1';
 END;
```

此例中的触发器 TRG_INS_DEL 既是 INSERT 触发器又是 DELETE 触发器，对基表 T1 的 INSERT 和 DELETE 操作都会激发该触发器。

2．元组级触发器和语句级触发器

根据触发器的级别可将触发器分为元组级触发器（也称行级触发器）和语句级触发器。

元组级触发器为触发命令所影响的每条记录激发一次。假如一个 DELETE 命令从表中删除了 1000 行记录，那么这个表上的元组级 DELETE 触发器将被执行 1000 次。元组级触发器常用于数据审计、完整性检查等应用中。元组级触发器是在触发器定义语句中通过 FOR EACH ROW 子句创建的。对于元组级触发器，可以用一个 WHEN 子句来限制针对当前记录是否执行该触发器。WHEN 子句包含一条布尔表达式，当它的值为 TRUE 时，执行触发器；否则，跳过该触发器。

语句级触发器对每个触发命令执行一次。例如，对于一条将 500 行记录插入表 TABLE_1 中的 INSERT 语句，这个表上的语句级 INSERT 触发器只执行一次。语句级触发器一般用于对表上执行的操作类型引入附加的安全措施。语句级触发器是在触发器定义语句中通过 FOR EACH STATEMENT 子句创建的，该子句可缺省。

以下分别是元组级触发器和语句级触发器的例子。

```
CREATE OR REPLACE TRIGGER TRG_INS_ROW
BEFORE INSERT ON T1
FOR EACH ROW
BEGIN
   PRINT 'INSERT' || :NEW.COL1|| ' ON T1';
END;

CREATE OR REPLACE TRIGGER TRG_DEL_ST
AFTER DELETE ON T1
FOR EACH STATEMENT        --  此子句可省略
BEGIN
   PRINT 'AFTER DELETE ON T1';
END;
```

3．BEFORE 和 AFTER 触发器

触发器服务于触发事件，通过在触发器定义语句中指定 BEFORE 或 AFTER 关键字可以选择在触发事件之前还是之后运行触发器。显然，BEFORE 触发器在触发事件之前执行，而 AFTER 触发器在触发事件之后执行。

在元组级触发器中可以引用当前修改的记录在修改前后的值，修改前的值称为旧值，修改后的值称为新值。对于插入操作不存在旧值，而对于删除操作则不存在新值。

对于新、旧值的访问请求常常决定一个触发器是 BEFORE 类型还是 AFTER 类型。如果需要通过触发器对插入的行设置列值，那么为了能设置新值，需要使用一个 BEFORE 触发器，因为在 AFTER 触发器中不允许用户设置已插入的值。在审计应用中则经常使用 AFTER 触发器，因为元组修改成功后才有必要运行触发器，而成功地完成修改意味着成功地通过了该表的引用完整性约束。

以下是 BEFORE 触发器和 AFTER 触发器的举例。这里假定 T1 是用户 USER1 建的基表。

```
CREATE OR REPLACE TRIGGER TRG_INS_BEFORE
    BEFORE INSERT ON T1
    FOR EACH ROW
    BEGIN
       :NEW.COL1:=:NEW.COL1+1;
    END;
```

该触发器在插入一条记录前，将记录中 COL1 列的值加 1。

```
CREATE OR REPLACE TRIGGER TRG_INS_AFTER
    AFTER INSERT ON T1
    FOR EACH ROW
    BEGIN
       INSERT INTO T_AUDIT VALUES(:NEW.COL1, 'INSERT ON T1');
    END;
```

该触发器在插入一条记录后，将插入的值以及操作类型记录到用于审计的表 T_AUDIT 中。

4．合法的触发器类型

综上所述，在一张基表上所允许的可能的触发器类型共有 12 种，如下表所示：

名　　称	功　　能
BEFORE INSERT	在一个 INSERT 处理前激发一次
BEFORE INSERT FOR EACH ROW	每条新记录插入前激发
AFTER INSERT	在一个 INSERT 处理后激发一次
AFTER INSERT FOR EACH ROW	每条新记录插入后激发
BEFORE DELETE	在一个 DELETE 处理前激发一次
BEFORE DELETE FOR EACH ROW	每条记录被删除前激发
AFTER DELETE	在一个 DELETE 处理后激发一次
AFTER DELETE FOR EACH ROW	每条记录被删除后激发
BEFORE UPDATE	在一个 UPDATE 处理前激发一次
BEFORE UPDATE FOR EACH ROW	每条记录被修改前激发
AFTER UPDATE	在一个 UPDATE 处理后激发一次
AFTER UPDATE FOR EACH ROW	每条记录被修改后激发

12.1.2　触发器激发顺序

触发器是在 DML 语句运行时激发的。下面是执行 DML 语句的算法步骤：

1．如果有语句级前触发器的话，先运行该触发器；

2．对于受语句影响每一行：

（1）如果有行级前触发器的话，运行该触发器；

（2）执行该语句本身；

（3）如果有行级后触发器的话，运行该触发器；

3．如果有语句级后触发器的话，运行该触发器。

为了说明上面的算法，假设我们创建表 students 并在其上创建了所有四种 UPDATE 触发器，即之前、之后、语句级和行级。其代码如下：

```
CREATE TABLE students (
      STUDENT_ID   INT PRIMARY KEY,
      NAME             VARCHAR(30),
      AGE          SMALLINT,
      GENDER       CHAR,
      MAJOR        VARCHAR(30)
    );

    INSERT INTO students VALUES(10, 'Bill', 19, 'M', 'Computer');
    INSERT INTO students VALUES(11, 'Susan', 18, 'F', 'History');
    INSERT INTO students VALUES(12, 'John', 19, 'M', 'Computer');

    CREATE OR REPLACE TRIGGER Students_Before_St
    BEFORE UPDATE ON students
    BEGIN
      PRINT 'BEFORE UPDATE TRIGGER FIRED';
    END;

    CREATE OR REPLACE TRIGGER Students_After_St
    AFTER UPDATE ON students
    BEGIN
      PRINT 'AFTER UPDATE TRIGGER FIRED';
    END;

    CREATE OR REPLACE TRIGGER Students_Before_Row
    BEFORE UPDATE ON students
    FOR EACH ROW
    BEGIN
```

```
    PRINT 'BEFORE UPDATE EACH ROW TRIGGER FIRED';
END;

CREATE OR REPLACE TRIGGER Students_After_Row
AFTER UPDATE ON students
FOR EACH ROW
BEGIN
    PRINT 'AFTER UPDATE EACH ROW TRIGGER FIRED';
END;
```

现在，执行更新语句：

```
UPDATE students SET AGE=AGE+1;
```

该语句对三行有影响。语句级前触发器和语句级后触发器将各自运行一次，而行级前触发器和行级后触发器则各运行三次。因此，服务器返回的打印消息应为：

BEFORE UPDATE TRIGGER FIRED
BEFORE UPDATE EACH ROW TRIGGER FIRED
AFTER UPDATE EACH ROW TRIGGER FIRED
BEFORE UPDATE EACH ROW TRIGGER FIRED
AFTER UPDATE EACH ROW TRIGGER FIRED
BEFORE UPDATE EACH ROW TRIGGER FIRED
AFTER UPDATE EACH ROW TRIGGER FIRED
AFTER UPDATE TRIGGER FIRED

同类触发器的激发顺序没有明确的定义。如果顺序非常重要的话，应该把所有的操作组合在一个触发器中。

12.1.3　新、旧行值的引用

前面曾经提到，在元组级触发器内部，可以访问正在处理中的记录的数据，这种访问是通过两个引用变量:OLD 和:NEW 实现的。:OLD 表示记录被处理前的值，:NEW 表示记录被处理后的值，标识符前面的冒号说明它们是宿主变量意义上的连接变量，而不是一般的 DMPL/SQL 变量。我们还可以通过引用子句为这两个行值重新命名。

引用变量与其它变量不在同一个命名空间，所以变量可以与引用变量同名。在触发器体中使用引用变量时，必须采用下列形式：

```
:引用变量名.列名
```

其中，列名必须是触发表中存在的列，否则编译器将报错。

下表总结了标识符:OLD 和:NEW 的含义。

触发语句	标识符:OLD	标识符:NEW
INSERT	无定义，所有字段都为 NULL	该语句结束时将插入的值
UPDATE	更新前行的旧值	该语句结束时将更新的值
DELETE	行删除前的旧值	无定义，所有字段都为 NULL

:OLD 引用变量只能读取，不能赋值（因为设置这个值是没有任何意义的）；而:NEW 引用变量则既可读取，又可赋值（当然必须在 BEFORE 类型的触发器中，因为数据操作完成后再设置这个值也是没有意义的）。通过修改:NEW 引用变量的值，我们可以影响插入或修改的数据。

注意：对于 INSERT 操作，引用变量:OLD 无意义；而对于 DELETE 操作，引用变量:NEW 无意义。如果在 INSERT 触发器体中引用:OLD，或者在 DELETE 触发器体中引用:NEW，不会产生编译错误。但是在执行时，对于 INSERT 操作，:OLD 引用变量的值为空值；对于 DELETE 操作，:NEW 引用变量的值为空值，且不允许被赋值。

下例中触发器 GenerateValue 使用了:NEW 引用变量。该触发器是一个 INSERT 前触发器，其目的是自动生成一些字段值。

```
CREATE TABLE Product_In_Depot (
   Product_ID         CHAR (12) NOT NULL ,
   Depot_ID        CHAR (12) NOT NULL ,
   Product_name    CHAR (24) NOT NULL ,
   Price           MONEY,
   Num             INT,
   SumMoney        MONEY,
   Operater        CHAR(12) NOT NULL,
   Remark          VARCHAR(128)
);

CREATE OR REPLACE TRIGGER GenerateValue
BEFORE INSERT ON Product_In_Depot
FOR EACH ROW
BEGIN
   :new.SumMoney:=:new.Price*:new.Num;
   :new.Operater:=USER;
END;
```

触发器 GenerateValue 实际上是修改引用变量:NEW 的值，这就是:NEW 引用变量的用途之一。当执行一个 INSERT 语句时，无论用户是否给定 SumMoney 和 Operater 字段的值，触发器都将自动利用 Price 字段和 Num 字段来计算 SumMoney，并使用当前用户名来填写 Operater 字段。例如，用户可以执行如下所示的 INSERT 语句：

```
INSERT INTO Product_In_Depot(Product_ID, Depot_ID, Product_name, Price, Num)
VALUES('p0001', 'd0002', 'shoes', 120.50, 100);
```

该语句可以正确执行。尽管我们没有为非空列 Operater 指定值，但触发器可以提供它。事实上，即使为 SumMoney 和 Operater 字段指定了值，该值也会被忽略，因为触发器将改变该值。如果用户 USERA 执行下面的命令：

```
INSERT INTO Product_In_Depot
VALUES('p0001', 'd0002', 'shoes', 120.50, 100, 0, 'Anonymous',NULL);
```

新插入元组的 SumMoney 字段将被触发器修改为 12050.00，Operater 字段将被修改为当前用户名‘USERA’，而不是语句中的 0 和‘Anonymous’。

12.1.4　触发器谓词

如前面介绍的，触发事件可以是多个数据操作的组合，即一个触发器可能既是 INSERT 触发器，又是 DELETE 或 UPDATE 触发器。当一个触发器可以为多个 DML 语句触发时，在这种触发器体内部可以使用三个谓词：INSERTING、DELETING 和 UPDATING 来确定当前执行的是何种操作。这三个谓词的含义分别为：

谓 词	状　态
INSERTING	当触发语句为 INSERT 时为真，否则为假
DELETING	当触发语句为 DELETE 时为真，否则为假
UPDATING	当触发语句为 UPDATE 时为真，否则为假

虽然在其他 DMPL/SQL 语句块中也可以使用这三个谓词，但这时它们的值都为假。

下例中的触发器 LogChanges 使用这三个谓词来记录表 students 发生的所有变化。除了记录这些信息外，它还记录对表进行变更的用户名。该触发器的记录存放在表 T_audit 中，其结构如下：

```
CREATE TABLE T_audit (
    change_type         CHAR NOT NULL,
    changed_by          VARCHAR(8) NOT NULL,
    op_timestamp    DATE NOT NULL,
    old_student_id INT,
    old_name        VARCHAR(30),
    old_age             SMALLINT,
    old_gender          CHAR,
    old_major       VARCHAR(30),
    new_student_id      INT,
    new_name            VARCHAR(30),
    new_age         SMALLINT,
    new_gender          CHAR,
    new_major           VARCHAR(30)
);
```

触发器 LogChanges 的创建语句如下：

```
CREATE OR REPLACE TRIGGER LogChanges
    AFTER INSERT OR DELETE OR UPDATE ON students
    FOR EACH ROW
    DECLARE
     v_ChangeType CHAR(1);
BEGIN
```

```
    /* 'I'表示 INSERT 操作，'D'表示 DELETE 操作，'U'表示 UPDATE 操作  */
    IF INSERTING THEN
      v_ChangeType := 'I';
    ELSIF UPDATING THEN
      v_ChangeType := 'U';
    ELSE
      v_ChangeType := 'D';
    END IF;
    /*  记录对 students 做的所有修改到表 T_audit 中，包括修改人和修改时间  */
    INSERT INTO T_audit
    VALUES
    (v_ChangeType, USER, SYSDATE,
    :old.student_id, :old.name, :old.age, :old.gender, :old.major,
    :new.student_id, :new.name, :new.age, :new.gender, :new.major);
END;
```

12.1.5　变异表

系统对于元组级触发器要访问的表有一些限制。为了定义这些限制，有必要来理解什么是变异表（MUTATING TABLE）。变异表就是当前被DML语句修改的表。对触发器来说，变异表就是创建该触发器的表。元组级触发器体中的SQL语句不能读或修改触发语句的变异表。

上述限制适用于所有的元组级触发器，但是有一种情况例外。如果INSERT语句为值插入的话，则在该行的之前和之后，触发器不把触发表作为变异表对待，这是在元组级触发器可能读入或修改触发表时的唯一案例。查询插入则总是把触发表作为变异表对待，即使其子查询仅返回一行也是如此。

下面是一个有关变异表的例子：

例 假设我们要把每个专业的学生名额限制在五个。我们可以通过使用表students上的元组级INSERT或UPDATE前触发器来实现这种限制，下面是该触发器的代码：

```
CREATE OR REPLACE TRIGGER LimitMajors
BEFORE INSERT OR UPDATE OF major ON students
FOR EACH ROW
DECLARE
    v_MaxStudents            SMALLINT := 5;
    v_CurrentStudents SMALLINT;
    Too_Many_Stu_In_Major     EXCEPTION FOR -10001;
BEGIN
    /*  计算本专业已有学生人数  */
    SELECT COUNT(*)
```

```
        INTO v_CurrentStudents
        FROM students
        WHERE major = :new.major;
        /*  如果已经没有名额，抛出异常  */
        IF v_CurrentStudents + 1 > v_MaxStudents THEN
            RAISE Too_Many_Stu_In_Major;
        END IF;
    END;
```

看上去该触发器好象可以实现需要的功能。然而，如果我们执行更新语句：

```
UPDATE students SET MAJOR = 'History' WHERE STUDENT_ID = 10;
```

服务器会报错：试图更新变异表。

这是由于触发器 LimitMajors 企图查询自己的触发表（该表是变异表）而导致的。因此，我们不能在该行级触发器中对表 students 进行查询，而只能在语句级触发器使用该查询语句。然而，我们还不能只是简单地把触发器 LimitMajors 修改为语句级触发器，这是因为我们需要在触发器体中使用:new.major 的值。解决该问题的方法是创建两个触发器，即一个行级触发器和一个语句级触发器。在行级触发器中，我们记录:new.major 的值，但不查询表 students；而查询任务由语句级触发器来实现并使用行触发器记录的值。为记录:new.major 的值，我们需要创建一张表 major_tmp：

```
CREATE TABLE major_tmp(major VARCHAR(30));
```

改进的触发器 RLimitMajors 和 SLimiMajors 的代码如下：

```
CREATE OR REPLACE TRIGGER RLimitMajors
BEFORE INSERT OR UPDATE OF major ON students
FOR EACH ROW
/*  行级触发器，记录修改后的元组的专业信息到表 major_tmp 中  */
BEGIN
    INSERT INTO major_tmp VALUES(:new.major);
END;

CREATE OR REPLACE TRIGGER SLimitMajors
AFTER INSERT OR UPDATE OF major ON students
DECLARE
    v_MaxStudents          SMALLINT := 5;
    v_CurrentStudents SMALLINT;
    v_major              VARCHAR(30);
    Too_Many_Stu_In_Major     EXCEPTION FOR -10001;
    C1 CURSOR FOR SELECT DISTINCT(MAJOR) FROM major_tmp;
BEGIN
    OPEN C1;
    LOOP
        FETCH C1 INTO v_major;
```

```
            EXIT WHEN C1%NOTFOUND;
            /* 计算本专业已有学生人数 */
            SELECT COUNT(*)
            INTO v_CurrentStudents
            FROM students
            WHERE major = v_major;
            /* 如果已经没有名额，抛出异常 */
            IF v_CurrentStudents > v_MaxStudents THEN
                RAISE Too_Many_Stu_In_Major;
            END IF;
        END LOOP;
        DELETE FROM major_tmp;
    END;
```

现在，我们向表 students 中插入数据直到出现了过多的历史专业为止，看看触发器能不能正确工作。

```
INSERT INTO students VALUES (3, 'Daniel', 20, 'M', 'History');
INSERT INTO students VALUES (4, 'Emily', 21, 'F', 'History');
INSERT INTO students VALUES (5, 'Ashley', 19, 'F', 'History');
INSERT INTO students VALUES (6, 'David', 18, 'M', 'History');
INSERT INTO students VALUES (7, 'James', 20, 'M', 'History');
```

在试图插入第 6 个历史专业的学生时，服务器将报错。

请注意在创建新触发器之前，一定要删除不正确的触发器 LimitMajors。

12.1.6　设计触发器的原则

在应用中使用触发器功能时，应遵循以下设计原则，以确保程序的正确和高效：

1．如果希望保证一个操作能引起一系列相关动作的执行，请使用触发器；

2．不要用触发器来重复实现 DM 中已有的功能。例如，如果用约束机制能完成希望的完整性检查，就不要使用触发器；

3．避免递归触发。所谓递归触发，就是触发器体内的语句又会激发该触发器，导致语句的执行无法终止。例如，在表 T1 上创建 BEFORE UPDATE 触发器，而该触发器中又有对表 T1 的 UPDATE 语句。

4．合理地控制触发器的大小和数目。要知道，一旦触发器被创建，任何用户在任何时间执行的相应操作都会导致触发器的执行，这将是一笔不小的开销。

12.2　触发器的删除

当用户需要从数据库中删除一个触发器时，可以使用触发器删除语句。其语法如下：

语法格式

```
DROP TRIGGER[ [<数据库名>.]<模式名>.]<触发器名>;
```

参数

1．<数据库名>　　指明被删除触发器所属的数据库，缺省为当前数据库。

2．<模式名>　　指明被删除触发器所属的模式。

3．<触发器名>　　指明被删除的触发器的名字。

使用说明

1．当数据库名缺省时，默认为删除当前数据库下的触发器，否则，应指明触发器所属的数据库。

2．当触发器的触发表被删除时，表上的触发器将被自动地删除。

3．除了 DBA 用户外，其他用户只能删除自己创建的触发器。

权限

执行该操作的用户必须是触发器的创建者，或者具有 DBA 系统权限。

举例说明

例 1　删除当前数据库下的触发器 TRG1。

```
DROP TRIGGER TRG1;
```

例 2　删除数据库 AAA，模式 SYSDBA 下的触发器 TRG2。

```
DROP TRIGGER AAA.SYSDBA.TRG2;
```

12.3　禁止和允许触发器

每个触发器创建成功后都自动处于允许状态（ENABLE），只要基表被修改，触发器就会被激发。但是在某些情况下，例如：

1．触发器体内引用的某个对象暂时不可用；

2．载入大量数据时，希望屏蔽触发器以提高执行速度；

3．重新载入数据。

用户可能希望触发器暂时不被触发，但是又不想删除这个触发器。这时，可将其设置为禁止状态（DISABLE）。

当触发器处于允许状态时，只要执行相应的 DML 语句，且触发条件计算为真，触发器体的代码就会被执行；当触发器处于禁止状态时，则在任何情况下触发器都不会被激发。根据不同的应用需要，用户可以使用触发器修改语句将触发器的状态设置为允许或禁止状态。其语法如下：

语法格式

```
ALTER TRIGGER [[<数据库名>.]<模式名>.]<触发器名> DISABLE | ENABLE;
```

参数

1．<数据库名>　　指明被修改的触发器所属的数据库。

2．<模式名>　　指明被修改的触发器所属的模式。

3．<触发器名>　　指明被修改的触发器的名字。

4．DISABLE　　指明将触发器设置为禁止状态。当触发器处于禁止状态时，在任何情况下触发器都不会被激发。

5．ENABLE　　　　指明将触发器设置为允许状态。当触发器处于允许状态时，只要执行相应的 DML 语句，且触发条件计算为真，触发器就会被激发。

使用说明

1．当数据库名缺省时，默认为修改当前数据库指定模式下的触发器，否则，应指明触发器所属的数据库。除了 DBA 用户外，其他用户只能修改自己创建的触发器。

2．触发器创建时，被默认设置为允许状态。

3．使用表修改语句可禁止或允许该表上所有的触发器。

权限

执行该操作的用户必须是触发器的创建者，或者具有 DBA 系统权限。

此外，还可以使用表修改语句来禁止或允许该表上所有的触发器。采用表修改语句来修改触发器状态不需要知道基表上触发器的名称。其语法如下：

```
<表修改语句> ::=
    ALTER TABLE [[<数据库名>.]<模式名>.]<表名>
    [其他功能子句……]
    {DISABLE | ENABLE} ALL TRIGGERS
```

下面这个例子中创建了一张基表和两个触发器。我们将通过触发器修改语句和表修改语句来禁止和允许触发器。

```
CREATE TABLE T1(
COL1    INT PRIMARY KEY,
COL2    VARCHAR(20));

CREATE OR REPLACE TRIGGER TRG_BI_T1
BEFORE INSERT ON T1
FOR EACH ROW
BEGIN
    PRINT ' TRG_BI_T1 IS FIRING';
END;

CREATE OR REPLACE TRIGGER TRG_AI_T1
AFTER INSERT ON T1
FOR EACH ROW
BEGIN
    PRINT ' TRG_AI_T1 IS FIRING';
END;
```

我们执行下列语句来禁止触发器 TRG_BI_T1 和 TRG_AI_T1：

```
ALTER TRIGGER TRG_BI_T1 DISABLE;
ALTER TRIGGER TRG_AI_T1 DISABLE;
```

也可以使用表修改语句来完成同样的任务：

```
ALTER TABLE T1 DISABLE ALL TRIGGERS;
```

向 T1 中插入一条数据：

```
INSERT INTO T1 VALUES(1, 'ABCDEFG');
```

服务器没有返回任何打印信息，因为表上的触发器已经被禁止了。下面我们重新允许这两个触发器。执行下列语句：

```
ALTER TRIGGER tRG_BI_T1 ENABLE;
ALTER TRIGGER tRG_AI_T1 ENABLE;
```

当然，我们也可以使用表修改语句：

```
ALTER TABLE T1 ENABLE ALL TRIGGERS;
```

现在，再向 T1 中插入一条数据：

```
INSERT INTO T1 VALUES(2, '1234567890');
```

由于激发了触发器 TRG_BI_T1 和 TRG_AI_T1，服务器返回打印消息：

```
TRG_BI_T1 IS FIRING
TRG_AI_T1 IS FIRING
```

12.4　触发器应用举例

正如我们在本章所介绍的，触发器是 DM 系统提供的重要机制。我们可以使用该机制来加强比正常的审计机制、完整性约束机制、安全机制等所能提供的功能更复杂的事务规则。为帮助用户更好地使用该机制，我们提供了一些触发器应用的例子供用户参考。

12.4.1　使用触发器实现审计功能

尽管 DM 系统本身已经提供了审计机制，但是在许多情况下我们还是可以利用触发器完成条件更加复杂的审计。与内置的审计机制相比，采用触发器实现的审计有如下优点：

1．使用触发器可针对更复杂的条件进行审计。例如：只对在每天 18:30 后修改表 IMPORTANT_INFO 的操作进行审计；

2．使用触发器不仅可以记录操作语句本身的信息，还可以记录被该语句修改的数据的具体值；

3．内置的审计机制将所有审计信息集中存放，而触发器实现的审计可针对不同的操作对象分别存放审计信息，便于分析。

虽然如此，触发器并不能取代内置的审计机制。因为内置审计机制的某些功能触发器是无法做到的。例如：

1．内置审计机制可审计的类型更多。触发器只能审计表上的 DML 操作，而内置审计机制可以针对各种操作、对象和用户进行审计；

2．触发器只能审计成功的操作，而内置审计机制能审计失败的操作；

3．内置审计机制使用起来更简单，并且其正确性更有保障。

用于审计的触发器通常都是 AFTER 类型。关于审计的实例，请参考第 12.1.4 节的例子，其中的触发器 LogChanges 就是一个典型的审计触发器。

12.4.2　使用触发器维护数据完整性

触发器与完整性约束机制都可以用于维护数据的完整性，但是二者之间存在着显著的区别。一般情况下，如果使用完整性约束机制可以完成约束检查，我们不建议用户使用触发器。这是因为：

（1）完整性约束机制能保证表上所有数据符合约束，即使是约束创建前存在的数据也必须如此；而触发器只保证其创建后的数据满足约束，但之前存在数据的完整性则得不到保证；

（2）完整性约束机制使用起来更简单，并且其正确性更有保障。

触发器通常用来实现完整性约束机制无法完成的约束检查和维护，例如：

1．引用完整性维护

删除被引用表中的数据时，级联删除引用表中引用该数据的记录；更新被引用表中的数据时，更新引用表中引用该数据的记录的相应字段。下例中，表 Dept_tab 为被引用表，其主关键字为 Deptno；表 Emp_tab 为引用表。其结构如下：

```
CREATE TABLE Dept_tab (
    Deptno    INT PRIMARY KEY,
    Dname     VARCHAR(15),
    Loc  VARCHAR(25)
);

CREATE TABLE Emp_tab (
    Empno     INT PRIMARY KEY,
    Ename     VARCHAR(15) NOT NULL,
    Job  VARCHAR(10),
    Sal   FLOAT,
    Deptno    INT
);

CREATE OR REPLACE TRIGGER Dept_del_upd_cascade
AFTER DELETE OR UPDATE OF Deptno ON Dept_tab
FOR EACH ROW
BEGIN
    IF DELETING THEN
        DELETE FROM Emp_tab
            WHERE Deptno = :old.Deptno;
    ELSE
        UPDATE Emp_tab SET Deptno = :new.Deptno
            WHERE Deptno = :old.Deptno;
    END IF;
```

```
END;
```

2．CHECK 规则检查

增加新员工或者调整员工工资时，保证其工资不超过规定的范围，并且涨幅不超过 25%。该例中，表 Emp_tab 记录员工信息；表 Salgrade 记录各个工种的工资范围，其结构如下：

```
CREATE TABLE Salgrade (
    Losal            FLOAT,        --最低工资
    Hisal       FLOAT,        --最高工资
    Job_classification   VARCHAR(10)     --工种编号
);

CREATE OR REPLACE TRIGGER Salary_check
BEFORE INSERT OR UPDATE OF Sal, Job ON Emp_tab
FOR EACH ROW
DECLARE
    Minsal          FLOAT;
    Maxsal          FLOAT;
    Salary_out_of_range      EXCEPTION FOR -10002;
BEGIN
   /* 取该员工所属工种的工资范围 */
   SELECT Losal, Hisal INTO Minsal, Maxsal FROM Salgrade
   WHERE Job_classification = :new.Job;
   /* 如果工资超出工资范围，报告异常 */
   IF (:new.Sal < Minsal OR :new.Sal > Maxsal) THEN
           RAISE Salary_out_of_range;
   END IF;
   /* 如果工资涨幅超出 25%，报告异常 */
   IF UPDATING AND (:new.Sal - :old.Sal) / :old.Sal > 0.25 THEN
           RAISE Salary_out_of_range;
   END IF;
END;
```

12.4.3　使用触发器保障数据安全性

在复杂的条件下，可以使用触发器来保障数据的安全性。同样，要注意不要用触发器来实现 DM 安全机制已提供的功能。使用触发器进行安全控制时，应使用语句级 BEFORE 类型的触发器，其优点如下：

（1）在执行触发事件之前进行安全检查，可以避免系统在触发语句不能通过安全检查的情况下做不必要的工作；

（2）使用语句级触发器，安全检查只需要对每个触发语句进行一次，而不必对语句影响的每一行都执行一次。

下面这个例子显示如何用触发器禁止在非工作时间内修改表 Emp_tab 中的工资（Sal）栏。非工作时间包括周末、公司规定的节假日以及下班后的时间。为此，我们还需要一个表 Company_holidays 来记录公司规定的节假日。其结构如下：

```
CREATE TABLE Company_holidays (
   Holiday    DATE
);
CREATE OR REPLACE TRIGGER Emp_permit_changes
BEFORE INSERT OR DELETE OR UPDATE OF SAL
ON Emp_tab
DECLARE
    Dummy            INTEGER;
    Invalid_Operate_time       EXCEPTION FOR -10002;
BEGIN
    /*  检查是否周末  */
    IF (DAYNAME(Sysdate) = 'Saturday' OR
        DAYNAME(Sysdate) = 'Sunday') THEN
        RAISE Invalid_Operate_time;
    END IF;
    /*  检查是否节假日  */
    SELECT COUNT(*) INTO Dummy FROM Company_holidays
        WHERE Holiday= Current_date;
    IF dummy > 0 THEN
        RAISE Invalid_Operate_time;
    END IF;
    /*  检查是否上班时间  */
    IF (EXTRACT(HOUR FROM Current_time) < 8 OR
         EXTRACT(HOUR FROM Current_time) >= 18) THEN
         RAISE Invalid_Operate_time;
    END IF;
END;
```

12.4.4　使用触发器派生字段值

触发器还经常用来自动生成某些字段的值，这些字段的值有时依赖于本记录中的其他字段的值，有时是为了避免用户直接对这些字段进行修改。这类触发器应该是元组级 BEFORE INSERT 或 UPDATE 触发器。因为：

（1）必须在 INSERT 或 UPDATE 操作执行之前生成字段的值；

（2）必须为每条元组自动生成一次字段的值。

关于使用触发器派生字段值的实例，请参考第 12.1.3 节中的例子。该例中使用触发器 GenerateValue 来自动产生商品的总价和操作员。

第 13 章　DM 安全管理

数据库的安全性是指保护数据库以防止不合法的使用所造成的数据泄露、更改或破坏。安全性问题不是数据库系统所独有的，计算机系统都有这个问题，只是在数据库系统中大量数据集中存放，而且为许多用户直接共享，是宝贵的信息资源，从而使安全性问题更为突出。

数据库安全性保护措施是否有效是衡量数据库系统的重要性能指标之一。DM 的安全管理就是为保护存储在 DM 数据库中的各类敏感数据的机密性、完整性和可用性提供必要的技术手段，防止对这些数据的非授权泄露、修改和破坏，并保证被授权用户能按其授权范围访问所需要的数据。

本章介绍的主要内容包括：角色与权限、自主存取控制、审计及安全检查。这些内容属于高级系统管理范畴，是系统管理员应该掌握的内容。安全管理对信息敏感部门尤为重要，应用系统设计者和系统管理员必须熟练掌握系统的安全特性，以便建立高安全性数据库应用系统。

虽然本章所讨论的内容是针对 DM 的，但其原理和方法与其它系统大同小异，因而，对于了解和掌握其它数据库管理系统的安全管理也是有帮助的。

13.1　创建角色语句

在 DM 系统中，可以对用户直接授权，也可以通过角色来授权，在实际的权限分配方案中，通常是用角色，先由 DBA 为数据库定义一系列的角色，然后再由 DBA 将权限分配给基于这些角色的用户。

角色可以分为系统级用户角色和一般用户角色。一般用户角色由 DBA 用户创建；系统级用户角色固定为四类：DBA、RESOURCE、AUDITOR 和 PUBLIC。

创建一般用户角色，该角色可以被授予某种客体的某种操作权限，其它用户或角色可以继承该角色的权限。

语法格式

```
CREATE ROLE  <角色名> [AT <数据库名>];
```

参数

<角色名>　　指明要创建的角色的名称。

<数据库名>　指明角色所属数据库的名称。

语句功能

供 DBA 用户在指定数据库中创建一般用户角色。

使用说明

1. 该语句的使用者需要具备 DBA 或者 CREATE ROLE 的权限，才能创建用户角色。
2. 若没有 AT <数据库名>指明角色所属的数据库，则角色属于当前数据库。

举例说明

例 在数据库 CW 中创建角色 DM_ROLE1，赋予表厂商登记的 SELECT 权限。

```
        CREATE ROLE DM_ROLE1 AT CW;
        GRANT SELECT ON 厂商登记 TO CW.DM_ROLE1;
```

注：创建角色、删除角色、对角色的授权、权限回收等对角色执行相关操作的 SQL 语句中，为了区分角色名的大小写，保留角色名的大小写形式，角色名可以一律加上双引号" "，比如创建角色 DM_ROLE1 的语句为：CREATE ROLE "DM_ROLE1"，用户 DM_USER1 继承 DM_ROLE1 权限的语句为：GRANT "DM_ROLE1" TO "DM_USER1"，如果不加双引号，角色名一律转换为大写。对角色的其他 SQL 语句，不重复说明。

13.2　删除角色语句

删除一个角色。

语法格式

```
DROP ROLE   [<数据库名>.]<角色名>
```

参数

<数据库名>　指明要删除的角色所属数据库的名称。

<角色名>　　指明要删除的角色的名称。

语句功能

供 DBA 用户在指定数据库中删除一般用户角色。

使用说明

该语句的使用者需要具备 DBA 的权限，角色删除才能成功。

举例说明

例 假定数据库 CW 中存在角色 DM_ROLE1，创建用户 RU，将角色权限授予 RU，回收角色权限，并删除角色：。

```
SET CURRENT DATABASE CW;
CREATE USER RU;
GRANT CW.DM_ROLE1 TO RU;
REVOKE CW.DM_ROLE1 FROM RU;
DROP ROLE CW.DM_ROLE1;
/*删除将会成功*/
```

13.3　授权语句(系统权限)

将指定的系统权限授予用户/角色。

语法格式

```
GRANT <特权>
        TO <接受者>{,<接受者>} | PUBLIC
        [WITH GRANT OPTION];

<特权> ::=    <动作> {, <动作>}
```

```
<动作> ::=    < CREATE  USER>|
         < CREATE  ROLE>|
         < CREATE  TABLE>|
         < CREATE  VIEW> |
         < CREATE  PROCEDURE>
         < CREATE  FUNCTION>|
         < CREATE  SCHEMA> |
         < CREATE  SEQUENCE>
<接受者>::=[<数据库名>.]<用户名> | [<数据库名>.]<角色名>
```

参数

1．<用户名>　　指明被授予权限的用户的名称。

2．<角色名>　　指明被授予权限的角色的名称。

3．PUBLIC　　将权限授予所有的用户。

4．WITH GRANT OPTION　　指明被授予者具有所授权限的转授权限。

使用说明

当指定了可选子句 WITH GRANT OPTION 时，接受者不仅具有该对象上给定权限的使用权，而且可以把这些权限转授给其他用户。

权限

授权者必须具有所授出权限的使用权和转授权。

举例说明

例 当前数据库中，系统管理员 SYSDBA 把建表和建视图的权限授给当前库中用户 RU。

```
GRANT CREATE TABLE, CREATE VIEW TO RU;
```

13.4 授权语句(实体权限)

将指定的实体的权限授予用户。

语法格式

```
GRANT <特权> ON  [<对象类型>] <对象名>
    TO <接受者>{,<接受者>} | PUBLIC
    [WITH GRANT OPTION];

<特权>::=    ALL [PRIVILEGES] | <动作> {, <动作>}

<动作>::=    INSERT | DELETE | EXECUTE |
         SELECT[(<列清单>)] |
         UPDATE[(<列清单>)] |
         REFERENCES[(<列清单>)]
```

```
<列清单>::= <列名> {,<列名>}

<对象类型>::=  DATABASE | FUNCTION | INDEX |
               PROCEDURE | TABLE | TRIGGER|
               VIEW | USER | ROLE| SCHEMA | SEQUENCE
<接受者>::=[<数据库名>.]<用户名> | [<数据库名>.]<角色名>
```

参数

1. ALL PRIVILEGES　授权者将其在被授权对象上具有的全部可转授权限授予被授权者。关键字 PRIVILEGES 可以缺省。

2．INSERT　授权者将表或视图上的 INSERT 权限授予被授权者。

3．DELETE　授权者将表或视图上的 DELETE 权限授予被授权者。

4．EXECUTE　授权者将存储过程或存储函数上的 EXECUTE 权限授予被授权者。

5．SELECT　授权者将表、视图或序列上的 SELECT 权限授予被授权者，<列清单>可指明被授予权限的表或视图的列。

6．UPDATE　授权者将表或视图上的 UPDATE 权限授予被授权者，<列清单>可指明被授予权限的表或视图的列。

7．REFERENCES　授权者将表上的 REFERENCES 权限授予被授权者，<列清单>可指明被授予权限的表的列。

8．PROCEDURE　指明被授权对象的类型为存储过程。

9．FUNCTION　指明被授权对象的类型为存储函数。

10．SEQUENCE　指明被授权对象的类型为序列。

11．<对象名>　指明被授权对象的名称，被授权对象可以是数据库、表、视图、索引、用户、角色、模式、存储过程、存储函数或序列。这里对象名必要时应使用“全名”，例如对于非当前数据库的对象，应加上数据库名前缀指明数据库，对非当前模式的模式对象，加上模式名前缀。缺省按当前数据库、当前模式处理。

使用说明

1．当授权者不是被授权对象的创建者时，语句中必须给定对象所属的模式名。权限接受者不能是 DBA、AUDITOR、被授权对象的创建者和授权者本身。当接受者指定为 PUBLIC 时，指的是数据库中可接受权限的所有合法用户。

2．由于 DBA 拥有所有的对象权限，因此在 DBA 角色的已授予对象权限节点下不显示任何对象权限

3．当指定了可选子句 WITH GRANT OPTION 时，接受者不仅具有该对象上给定权限的使用权，而且可以把这些权限转授给其他用户。

4．当多个用户对同一用户授予同一对象上的不同权限时，该用户在此对象上拥有的权限是他们的和。当多个用户对同一用户授同一种权限时，DM 系统并不重复登记，只登记最先的一个。

5．当多次给同一用户授予同一表上不同列的 SELECT、UPDATE 或 REFERENCES 权限时，用户得到的是这些列级权限的和，但不重复。

6．当用户具有表或视图上的 SELECT 或 UPDATE 权限时，如再对该用户在该表或

视图上授某些列的 SELECT 或 UPDATE 权限是无意义的，系统不作该权限的授权记录。

7. 当用户具有表或视图上部分列的 SELECT 或 UPDATE 权限时，再对该用户授此表或视图的表级 SELECT 或 UPDATE 权限，该用户具有此表或视图上的 SELECT 或 UPDATE 权限。

权限

授权者必须具有被授权对象上所授出权限的使用权和转授权，这对 DBA 和对象的创建者是隐含具有的。

举例说明

例 1 当前数据库为 CW，表的创建者 USER1 把所创建的厂商登记表的全部权限授给 CW 的用户 RU。

```
GRANT SELECT, INSERT, DELETE, UPDATE, REFERENCES
    ON 厂商登记
    TO RU;
该语句也可写为以下形式：
GRANT ALL PRIVILEGES ON 厂商登记 TO RU;
或
    GRANT ALL PRIVILEGES ON CW.USER1.厂商登记 TO CW.RU;
```

例 2 当前数据库为 CW，数据库管理员 DU 把当前库用户 RU 创建的存储过程 PROC1 的执行权 EXECUTE 授给用户 CU2，并使其具有该权限的转授权。

```
GRANT EXECUTE
    ON PROCEDURE RU.PROC1
    TO CU2
    WITH GRANT OPTION;
```

例 3 假定当前数据库中 RU 是表商品登记的创建者，用户 U11、U12、U13 在数据库中存在，且都是 DBA 系统权限用户。

```
（1）以 RU 身份登录，并执行语句：
    GRANT SELECT ON 商品登记 TO U11 WITH GRANT OPTION;
    /*正确*/
（2）以 U11 身份登录，并执行语句：
    GRANT SELECT ON 商品登记 TO U12 WITH GRANT OPTION;
    /*错误，表名前面缺少前缀 RU*/
    GRANT SELECT ON RU.商品登记 TO U12 WITH GRANT OPTION;
    /*正确*/
（3）以 U12 身份登录，并执行语句：
    GRANT SELECT ON 商品登记 TO U13 WITH GRANT OPTION;
    /*错误，表名前面缺少前缀 RU*/
    GRANT SELECT ON RU.商品登记 TO U13 WITH GRANT OPTION;
    /*正确*/
```

例 4 假定当前数据库中 RU 是表商品登记的创建者，用户 U21、U22、U23 在数据库中存在，且都不是 DBA 系统权限用户。

```
（1）以 RU 身份登录，并执行语句：
    GRANT SELECT ON  商品登记  TO U21 WITH GRANT OPTION;
    /*正确*/
（2）以 U21 身份登录，并执行语句：
    GRANT SELECT ON RU.商品登记  TO U22;
    /*正确*/
（3）以 U22 身份登录，并执行语句：
    GRANT SELECT ON RU.商品登记  TO U23;
    /*错误，用户 U22 没有 SELECT 权限的转授权*/
```

例 5　假定当前数据库中用户 RU 创建了一个名为商品登记表的基表，要求对于该表，既允许他的部下 CU1 能够进行查询、插入、删除和更新操作，又对其可操作的数据范围有所限制(比如价格大于等于 100 元和小于 1000 元的商品才能进行上述操作)。与用户 CU1 同组的操作员 CU2 和 CU3 可进行同样的工作，但要求其操作权限由用户 CU1 控制。

```
（1）以用户 RU 的身份登录：
    CREATE VIEW V2 AS
        SELECT *
        FROM  商品登记
        WHERE  价格>=100 AND  价格<1000 WITH CHECK OPTION;
    GRANT SELECT, INSERT, DELETE, UPDATE
        ON V2
        TO CU1
        WITH GRANT OPTION;
（2）以用户 CU1 的身份登录：
    GRANT SELECT, INSERT, DELETE, UPDATE ON RU.V2 TO CU2, CU3;
```

例 6　假定当前数据库中表的创建者 RU 把所创建的商品登记表的部分列的更新权限授予用户 CU1，CU1 再将此更新权限转授 U11。

```
（1）以 RU 的身份登录，并执行语句：
    GRANT UPDATE (商品编号, 商品名, 价格)
        ON  商品登记
        TO CU1
        WITH GRANT OPTION;
（2）以 CU1 的身份登录，并执行语句，把列级更新权限授给用户 U11：
    GRANT UPDATE(商品编号, 商品名, 价格)
        ON RU.商品登记
        TO U11;
```

例 7 假定当前数据库中表 T1 的拥有者 RU 将该表上的所有权限及其转授权授予角色 ROLE_A。假设 CU1 为角色 ROLE_A 中的成员，把 T1 上的 SELECT 权限转授给 CU2。

```
（1）以 RU 的身份登录，并执行语句：
    GRANT ALL ON T1 TO ROLE_A WITH GRANT OPTION;
```

（2）以 CU1 的身份登录，并执行语句：

```
GRANT SELECT ON RU.T1 TO CU2;
```

13.5 授权语句(角色权限)

一般用户角色包含用户或其它角色，也就是用户或其它角色可以继承该角色的权限。

语法格式

```
GRANT <角色名>, { <角色名>}
      TO  <角色名或用户名>,{<角色名或用户名>}
      [WITH GRANT OPTION];
<角色名或用户名>::=[<数据库名>.]<用户名> | [<数据库名>.]<角色名>
```

参数

1．<角色名>　　　　指明系统中已经创建的一个角色的名称。

2．<角色名或用户名>　指明要继承指定角色权限的其它角色或用户。

使用说明

角色授权不允许出现循环。

权限

授权者必须具有所授出权限的使用权和转授权。

举例说明

例 假设当前数据库中存在角色 DM_ROLE1，角色 DM_ROLE2，用户 DM_USER1。

（1）让用户 DM_USER1 继承角色 DM_ROLE1 的权限：

```
GRANT DM_ROLE1 TO DM_USER1;
```

（2）让角色 DM_ROLE2 继承角色 DM_ROLE1 的权限：

```
GRANT DM_ROLE1 TO DM_ROLE2;
```

13.6 回收权限语句(系统权限)

回收用户或者角色的系统权限。

语法格式

```
REVOKE <特权>
FROM <被回收者>{,<被回收者>};

<特权>::=      <动作> {, <动作>}
<动作>::=      < CREATE   USER> |
            < CREATE   ROLE> |
            < CREATE   TABLE> |
            < CREATE   VIEW> |
            < CREATE   PROCEDURE> |
            < CREATE   FUNCTION> |
```

```
        < CREATE   SCHEMA> |
        < CREATE   SEQUENCE>
<被回收者>::=[<数据库名>.]<用户名> | [<数据库名>.]<角色名>
```

参数

<数据库名> 指明被回收权限的用户或角色所属数据库名称

<用户名> 指明被回收权限的用户的名称

<角色名> 指明被回收权限的角色的名称。

权限

执行本操作的用户必须是所回收权限的直接或间接授予者，或者拥有 DBA 系统权限。

举例说明

例 假设当前数据库中系统管理员 SYSDBA 把建表、建视图的权限授给了用户 CU。现在，系统管理员 SYSDBA 把用户 CU 的建表权限收回：

```
REVOKE CREATE TABLE FROM CU;
```

13.7 回收权限语句(实体权限)

回收用户对指定实体的权限。

语法格式

```
REVOKE <特权>
ON [<对象类型>]<对象名>
FROM <被回收者> {,<被回收者>};

<特权>::=      ALL [PRIVILEGES] | <动作> {, <动作>}

<动作>::=      INSERT | DELETE | EXECUTE |
               SELECT[(<列清单>)] |
               UPDATE[(<列清单>)] |
               REFERENCES[(<列清单>)]

<列清单>::=   <列名> {, <列名>}

<对象类型>::=   DATABASE | FUNCTION | INDEX |
                  PROCEDURE | TABLE | TRIGGER|
                  VIEW | USER | ROLE| SCHEMA | SEQUENCE
```

参数

1. <对象名>　　指明被回收权限的对象的名称，对象可以是表、视图、存储过程、存储函数或序列等。这里对象名必要时应使用“全名”，例如对于非当前数据库的对象，应加上数据库名前缀指明所属数据库，对非当前模式的模式对象，加上模式名前缀。缺

省按当前数据库、当前模式处理。

2．<被回收者>　指明被回收权限的用户或角色的名称。

注：其他参数参见 13.4 节。

使用说明

1．不是对象的创建者时，语句中必须给定对象所属的模式名。

2．必须是数据库中已存在的用户。被回收权限必须是被回收者当前部分或全部拥有的权限。

3．执行该语句，除回收了被回收者的权限外，同时还回收了被回收者曾经转授出去的相应权限。这就是权限的级联回收。

4．回收用户的实体级权限时，连同相应的转授权一并回收。操作权限可以独立存在，而转授权不能够独立存在。

权限

执行本操作的用户必须是给定对象上所回收权限的直接或间接授予者，或者拥有 DBA 系统权限。

举例说明

例 1 当前数据库中视图 VIEW1 的创建者 RU 从用户 CU 处回收其对视图 VIEW1 的所有权限。

```
REVOKE ALL ON VIEW1 FROM CU;
```

例 2 当前数据库中 DBA 用户 DU 从用户 CU 处回收其对存储函数 FUNC1 的 EXECUTE 权限，该存储函数的创建者为用户 RU。

```
REVOKE EXECUTE ON FUNCTION RU.FUNC1 FROM CU;
```

例 3 假定当前数据库中用户 RU 是表商品登记的创建者，用户 U11、U12、U13 为新建的 DBA 用户。本例说明权限的级联回收。

```
（1）以 RU 身份登录，并执行语句：
    GRANT SELECT, INSERT ON 商品登记 TO U11 WITH GRANT OPTION;
（2）以 U11 身份登录，并执行语句：
    GRANT SELECT ON RU.商品登记 TO U12 WITH GRANT OPTION;
（3）以 U12 身份登录，并执行语句：
    GRANT SELECT ON RU.商品登记 TO U13 WITH GRANT OPTION;
（4）以 RU 身份登录，并执行语句：
    REVOKE SELECT ON 商品登记 FROM U11;
```

执行的结果为：用户 U11 在商品登记表上有插入操作权及插入操作转授权；用户 U12 和 U13 在商品登记表上没有权限。

例 4 假定当前数据库中用户 RU 是表商品登记的创建者，用户 U21、U22、U23 为新建的 DBA 用户。本例说明非 DBA 用户的收权者只能回收由自己直接或间接授予被收权者的权限，而不能回收其他用户授予的权限。

```
（1）以 RU 身份登录，并执行语句：
    GRANT ALL PRIVILEGES ON 商品登记 TO U21 WITH GRANT OPTION;
    GRANT INSERT, UPDATE ON 商品登记 TO U22, U23;
（2）以 U21 身份登录，并执行语句：
```

```
GRANT SELECT ON RU.商品登记  TO U22 WITH GRANT OPTION;
GRANT SELECT ON RU.商品登记  TO U23 WITH GRANT OPTION;
（3）以 U21 身份登录，并执行语句：
REVOKE ALL PRIVILEGES ON RU.商品登记  FROM U22,U23;
```

执行的结果为：用户 U21 在商品登记表上有全部操作权及全部转授权；用户 U22 和 U23 在该表上仍然具有插入权(INSERT)和更新权(UPDATE)。

13.8　回收权限语句(角色权限)

用来回收用户或其它角色从指定角色继承过来的权限

语法格式

```
REVOKE   [<数据库名>.]<角色名> FROM   <角色名或用户名>;
<角色名或用户名> ::=[<数据库名>.]<用户名> | [<数据库名>.]<角色名>
```

参数

1．<角色名>　　　　指明系统中已经创建的一个角色的名称。

2．<角色名或用户名>　指明要继承指定角色权限的其它角色或用户。

权限

该语句的使用者需要具备 DBA 的权限。

举例说明

例 假定当前数据库中存在角色 DM_ROLE1，角色 DM_ROLE2，用户 DM_USER1。

```
（1）让用户 DM_USER1 继承角色 DM_ROLE1 的权限：
   GRANT DM_ROLE1 TO DM_USER1;
（2）回收用户 DM_USER1 从角色 DM_ROLE1 继承过来的权限：
   REVOKE DM_ROLE1 FROM   DM_USER1;
（3）让角色 DM_ROLE2 继承角色 DM_ROLE1 的权限：
   GRANT DM_ROLE1 TO DM_ROLE2;
（4）回收角色 DM_ROLE2 从角色 DM_ROLE1 继承过来的权限：
   REVOKE DM_ROLE1 FROM   DM_ROLE2;
```

13.9　审计设置语句

数据库审计员指定被审计对象的活动称为审计设置。DM 提供审计设置语句来实现这种设置，被审计的对象可以是某类操作，也可以是某些用户在数据库中的全部行踪。只有预先设置的操作和用户才能被 DM 系统自动进行审计。DM 的审计设置是立即生效的，不需系统重新启动。

在 DM 系统中，专门为审计设置了开关：在系统运行的过程中，用户也可通过调用系统函数 SET_AUDIT 来设置开关。进行审计设置时，审计开关必须打开。

语法格式

```
AUDIT <审计操作类型>[,<审计操作类型>]
```

```
        [ON <表名视图名定义>] [BY <用户名定义> {,<用户名定义>}]
        [WHENEVER SUCCESSFUL|WHENEVER NOT SUCCESSFUL]

    <表名视图名定义> ::=[[<数据库名>.] <模式名>.]<表或视图名>
    <表或视图名>::=<基表名>|<视图名>
    <用户名定义>::=[<数据库名>.]<用户名>
```

参数

<审计操作类型>包括下列类型：

ALL 所有可审计操作；
ALTER CONTEXT INDEX 修改全文索引
ALTER DATABASE 修改数据库操作
ALTER LOGIN 修改登录操作
ALTER TABLE 修改表操作；
ALTER TRIGGER 修改触发器操作；
AUDIT 审计操作；
CALL 调用存储过程操作；
COMMIT 提交事务操作；
CONNECT LOGIN 和 LOGOUT 操作；
CREATE CONTEXT INDEX 创建全文索引
CREATE DATABASE 创建数据库操作；
CREATE FUNCTION 创建函数操作；
CREATE INDEX 创建索引操作；
CREATE LOGIN 创建登录
CREATE PROCEDURE 创建存储过程操作；
CREATE ROLE 创建角色操作；
CREATE SCHEMA 创建模式操作；
CREATE SEQUENCE 创建序列
CREATE TABLE 创建表操作；
CREATE TRIGGER 创建触发器操作；
CREATE USER 创建用户
CREATE VIEW 创建视图操作；
DROP CONTEXT INDEX 删除全文索引
DROP DATABASE 删除数据库操作
DROP FUNCTION 删除函数操作；
DROP INDEX 删除索引操作；
DROP LOGIN 删除登录操作
DROP PROCEDURE 删除存储过程操作；
DROP ROLE 删除角色操作；
DROP SCHEMA 删除模式操作；
DROP SEQUENCE 删除序列

DROP TABLE　　　　删除表操作；
DROP TRIGGER　　　　删除触发器操作；
DROP USER　　　　删除用户操作；
DROP VIEW　　　　删除视图；
GRANT　　　　授权操作；
LOCK TABLE　　　　锁表操作；
NOAUDIT　　　　取消审计设置操作；
REVOKE　　　　取消权限操作；
ROLLBACK　　　　回滚操作；
INSERT　　　　插入操作；
SELECT　　　　查询操作；
DELETE　　　　删除操作；
UPDATE　　　　更新操作；

其中，INSERT、SELECT、DELETE 和 UPDATE 操作必须指定所操作的表或视图。

举例说明

例 1 通过系统级审计设置使得进入和退出系统操作不成功完成时予以审计。

```
AUDIT CONNECT WHENEVER NOT SUCCESSFUL;
```

例 2 对商品登记表进行的所有不成功操作均予以审计。

```
AUDIT ALL ON CW.RU.商品登记 WHENEVER NOT SUCCESSFUL;
```

例 3 对厂商登记表进行的查询、插入、删除和更新操作，不管是否成功均予以审计。

```
AUDIT SELECT, INSERT, DELETE, UPDATE ON CW.RU.厂商登记;
```

例 4 对数据库 CW 的用户 RU 进行的所有不成功操作均予以审计。

```
AUDIT ALL BY CW.RU WHENEVER NOT SUCCESSFUL;
```

例 5 对数据库 CW 的用户 CU1、CU2 和 CU3 进行的操作，不管是否成功均予以审计。

```
AUDIT ALL BY CW.CU1, CW.CU2, CW.CU3;
```

13.10　审计取消语句

当数据库审计员认为对某些操作或某些用户不必再进行审计，可用 DM 提供的取消审计设置语句将原来对某些操作或用户的审计设置清除掉。同样，取消审计设置也是立即生效的，不需系统重新启动。

语法格式

```
NOAUDIT <审计操作类型>[,<审计操作类型>]
    [ON <表名视图名定义>] [BY <用户名定义> {,<用户名定义>}]
    [WHENEVER SUCCESSFUL|WHENEVER NOT SUCCESSFUL]
<表名视图名定义> ::=[[<数据库名>.] <模式名>.]<表或视图名>
<表或视图名>::=<基表名>|<视图名>
<用户名定义>::=[<数据库名>.]<用户名>
```

参数

参数说明参见 13.10 节。

举例说明

例 1 取消对进入和退出系统操作不成功的审计。

```
NOAUDIT CONNECT WHENEVER NOT SUCCESSFUL;
```

例 2 取消对商品登记表进行的所有不成功操作的审计。

```
NOAUDIT ALL ON CW.RU.商品登记 WHENEVER NOT SUCCESSFUL;
```

例 3 取消对厂商登记表的查询、插入、删除和更新操作的审计。

```
NOAUDIT SELECT, INSERT, DELETE, UPDATE ON CW.RU.厂商登记;
```

例 4 取消对数据库 CW 的用户 RU 的所有不成功操作的审计。

```
NOAUDIT ALL BY CW.RU WHENEVER NOT SUCCESSFUL;
```

例 5 取消对数据库 CW 的用户 CU1、CU2 和 CU3 的操作的审计。

```
NOAUDIT ALL BY CW.CU1, CW.CU2, CW.CU3;
```

13.11　审计信息查阅语句

当使用 DM 提供的审计机制进行了审计设置后，这些审计设置信息都记录在数据字典中。只要 DM 系统处于审计活动状态，系统则按审计设置进行审计活动，并将审计信息写入审计信息表。审计信息表内容包括操作者的用户名、所在数据库、所进行的操作、操作的对象、操作时间、当前审计条件等。具有 AUDITOR 权限的用户可以使用数据字典查询语句查询审计设置信息及审计信息表中记录的审计信息。

1.查询系统级审计设置信息

要查询系统级审计设置信息，语句如下：

```
SELECT * FROM SYSTEM.SYSAUDITOR.SYSAUDIT;
```

2.查询审计信息

要查询记录的所有审计信息，语句如下：

```
SELECT * FROM SYSTEM.SYSAUDITOR.SYSAUDITRECORDS;
```

数据字典表 SYSAUDIT 和 SYSAUDITRECORDS 的字典结构请参见第 9 章——数据库元信息。

附录 1　DM 保留字

A

ABSOLUTE, ACTION, ADD, AFTER, ALL, ALTER, AND, ANY, AS, ASC, ASSIGN, AUDIT, AUTHORIZATION, AVG

B

BACKUP, BEFORE, BEGIN, BETWEEN, BITMAP, BOTH, BY

C

CACHE, CALL, CASCADE, CASE, CAST, CATALOG, CHAIN, CHECK, CLOSE, CLUSTER, COALESCE, COLUMN, COMMIT, COMMITTED, COMMITWORK, COMPILE, CONNECT, CONSTRAINT, CONTAINS, CONTEXT, CONVERT, COUNT, CREATE, CROSS, CURRENT, CURSOR, CYCLE

D

DATABASE, DATAFILE, DATEADD, DATEDIFF, DATEPART, DEBUG, DECLARE, DECODE, DEFAULT, DEFERRABLE, DELETE, DELETING, DESC, DISCONNECT, DISABLE, DISTINCT, DROP

E

EACH, ELSE, ELSEIF, ELSIF, ENABLE, ENCRYPTION, END, ESCAPE, EXCEPTION, EXCLUSIVE, EXECUTE, EXEC, EXISTS, EXIT, EXPLAIN, EXTERNAL, EXTRACT

F

FALSE, FETCH, FILLFACTOR, FILEGROUP, FIRST, FOR, FOREIGN, FOUND, FROM, FULL, FUNCTION

G

GLOBAL, GOTO, GRANT, GROUP

H

HAVING, HEXTORAW

I

IDENTIFIED, IDENTITY, IDENTITY_INSERT, IDENTITY, IF, IFNULL, IMMEDIATE, IN, INCREASE, INCREMENT, INDEX, INITIAL, INITIALLY, INNER, INSERT, INSERTING, INTENT, INTERVAL, INTO, IS, ISNULL, ISOLATION, ISOPEN

J

JOIN

K

KEY

L

LAST, LEADING, LEFT, LEVEL, LIKE, LOCAL, LOCK, LOGIN, LOOP

M

MATCH, MAX, MAXVALUE, MIN, MINEXTENTS, MINVALUE, MODE, MODIFY

N

NATURAL, NEW, NEXT, NOAUDIT, NOT, NOCACHE, NOCYCLE, NOMAXVALUE, NOMINVALUE, NOORDER, NOTFOUND, NOWAIT, NULL, NULLIF, NVL

O

OF, OFF, OLD, ON, ONLY, OPEN, OPTION, OR, ORDER, OUT, OUTER, OVERLAPS

P

PARTIAL, PENDANT, PERCENT, PRECISION, PRESERVE, PRIMARY, PRINT, PRIOR, PRIVILEGES, PROCEDURE

R

RAISE, RAWTOHEX, READ, REBUILD, REFERENCES, REFERENCING, RELATIVE, RENAME, REPEAT, REPEATABLE, REPLACE, RESTORE, RESTRICT, RETURN, REVERSE, REVOKE, RIGHT, ROLE, ROLLBACK, ROW, ROWCOUNT, ROWNUM, ROWS

S

SAVEPOINT, SCHEMA, SELECT, SEQUENCE, SERIALIZABLE,SET, SHARE, SOME, SQL, STATEMENT, STORAGE, SUBSTRING, SUCCESSFUL, SUM, SYNC

T

TABLE, TEMPORARY, THEN, TIES, TIMESTAMPADD, TIMESTAMPDIFF, TO, TOP, TRAILING, TRANSACTION, TRIGGER, TRIGGERS, TRIM, TRUE, TRUNCATE

U

UNCOMMITTED, UNION, UNIQUE, UNTIL, UPDATE, UPDATING, USER, USING

V

VALUES, VARYING, VIEW, VSIZE

W

WHEN, WHENEVER, WHERE, WHILE, WITH, WORK, WRITE

%TOOMANYROWS

附录 2　SQL 语法描述说明

< >　　表示一个语法对象。

::=　　定义符，用来定义一个语法对象。定义符左边为语法对象，右边为相应的语法描述。

|　　或者符，或者符限定的语法选项在实际的语句中只能出现一个。

{ }　　大括号指明大括号内的语法选项在实际的语句中可以出现 0…N 次（N 为大于 0 的自然数），但是大括号本身不能出现在语句中。

[ ]　　中括号指明中括号内的语法选项在实际的语句中可以出现 0…1 次，但是中括号本身不能出现在语句中。

关键字　关键字在 DM_SQL 语言中具有特殊意义，在 SQL 语法描述中，关键字以大写形式出现。但在实际书写 SQL 语句时，关键字可以为大写也可以为小写。

附录 3　SQL 命令参考

附录 4　系统存储过程和函数

一、系统存储过程（按功能分类）：

1）表复制功能：

1．SP_CREATE_DUPLICATE_CONNECT

定义：

```
SP_CREATE_DUPLICATE_CONNECT (
    slave_server varchar(128),
    port_num int,
    ret out int
)
```

功能说明：

连接到表复制的从服务器

入口参数：

slave_server：从服务器名

port_num：端口号

出口参数：

ret：失败返回 0，成功返回 1

2．SP_CREATE_DUPLICATE_TABLE

定义：

```
SP_CREATE_DUPLICATE_TABLE(
    m_tab varchar(500),
    s_server varchar(128),
    s_tab varchar(500),
    ret out int
)
```

功能说明：

在从服务器上创建指定表名的复制表

入口参数：

m_tab：主表名

s_server：从服务器名

s_tab：从表名

出口参数：

ret：失败返回 0，成功返回 1

3．SP_DESTROY_DUPLICATE_CONNECT

定义：

```
SP_DESTROY_DUPLICATE_CONNECT (
```

```
        slave_server varchar(128),
        ret out int
    )
```

功能说明：

删除与从服务器的连接

入口参数：

slave_server：从服务器名

出口参数：

ret：失败返回 0,成功返回 1

4．SP_DESTROY_DUPLICATE_TABLE

定义：

```
    SP_DESTROY_DUPLICATE_TABLE (
        m_tab varchar(500),
        s_server varchar(128),
        s_tab varchar(500),
        ret out int
    )
```

功能说明：

删除从服务器上的从表。

入口参数：

m_tab：主表名

s_server：从服务器名

s_tab：从表名

出口参数：

ret：失败返回 0，成功返回 1

5．SP_SNAP_DUPLICATE_TABLE

定义：

```
    SP_SNAP_DUPLICATE_TABLE(
        m_tab varchar(500),
        s_server varchar(128),
        s_tab varchar(500),
        ret out int
    )
```

功能说明：

复制主表数据到从表

入口参数：

m_tab ：主表名

s_server：从服务器名

s_tab：从表名

出口参数：

ret：失败返回 0，成功返回 1

2）数据重组功能：

1．SP_DB_REORGANIZE

定义：

```
SP_DB_REORGANIZE (
    dbname varchar(128)
)
```

功能说明：

对某个数据库中的所有对象在空间上进行重组

入口参数：

dbname：数据库名

出口参数；

无

2．SP_MERGE_TABLE

定义：

```
SP_MERGE_TABLE(
    dbname varchar(128),
    schname varchar(128),
    tablename varchar(128)
)
```

功能说明：

对指定表进行空间整理

入口参数：

dbname：db 名

schemaname：schema 名

tablename：表名

出口参数：

无

3．SP_MERGE_INDEX

定义：

```
SP_MERGE_INDEX (
    dbname varchar(128),
    schname varchar(128),
    indexname varchar(128)
)
```

功能说明：

对指定索引进行空间整理

入口参数：

dbname：db 名

schemaname：schema 名
indexname：索引名
出口参数：
无

4．SP_REORGANIZE_INDEX

定义：

```
SP_REORGANIZE_INDEX(
    dbname varchar(128),
    schname varchar(128),
    indexname varchar(128)
)
```

功能说明：
索引重组。
入口参数：
dbname：数据库名
schemaname：模式名
indexname：索引名
出口参数：
无

3）备份恢复功能：

1．SP_GET_BAK_INFO

定义：

```
sp_get_bak_info(
    db_name varchar(128)
)
```

功能说明：
将该数据库的所有备份信息都插入数据库一表中
入口参数：
db_name：数据库名
出口参数：
无

4）信息查询与修改功能：

1．SP_GET_DB_TAB_COL_ID

定义：

```
SP_GET_DB_TAB_COL_ID (
    dbname varchar(1000),
    objname varchar(1000),
    colname varchar(1000),
```

dbid out int,
objid out int,
colid out int
)

功能说明：

取得对象在数据库中的 id。

入口参数：

dbname：db 名
objname：表、视图名
colname：列名

出口参数：

dbid：dbid
objid：对象 id
colid：列 id

2．SP_GET_TABLE_INFO

定义：

SP_GET_TABLE_INFO(
dbname varchar(128),
schemaname varchar(128),
tablename varchar(128),
extents out int,
pages out int,
used_pages out int)

功能说明：

取得表的存储信息。

入口参数：

dbname：db 名
schemaname：schema 名
tablename：表名

出口参数：

extents：簇数。
pages：块数。
used_pages：已用页数

3．SP_MODIFY_MIN_EXTENTS

定义：

SP_MODIFY_MIN_EXTENTS(
dbname varchar(128),
schname varchar(128),
indexname varchar(128),
min_extents int

)

功能说明：

修改索引 minimal extent 参数

入口参数：

dbname：db 名

schemaname：schema 名

indexname：索引名。

min_extents：min_extents 值

出口参数：

无

4．SP_MODIFY_NEXT_EXTENTS

定义：

```
SP_MODIFY_NEXT_EXTENTS(
    dbname varchar(128),
    schname varchar(128),
    indexname varchar(128),
    next_extents int
)
```

功能说明：

修改索引 next_extentst 参数

入口参数：

dbname：db 名

schemaname：schema 名

indexname：索引名。

next_extents：next_extents 值

出口参数：

无

5．SP_UPDATE_STATISTICS

定义：

```
SP_UPDATE_STATISTICS（
    dbname varchar(128)
)
```

功能说明：

更新 database 统计信息。

入口参数：

dbname：数据库名

5）自定义表锁策略：

1．SP_SET_TABLE_OPTION

定义：

```
SP_SET_TABLE_OPTION(
    name varchar(128),
    option_name varchar(128),
    value int
)
```

功能说明：

设置表上锁策略

入口参数：

name：表名

option_name：设置条件，取值如下：

disallowpagelocks：禁止页锁

disallowrowlocks：禁止读写锁。

printable：可打印标记

value：1 表示 true，0 表示 false

出口参数：

无

二、系统函数：

1）备份恢复功能：

1．SF_BAK

定义：

```
int sf_bak(
    ddb_name varchar(128),
    name varchar(128),
    bdesc varchar(260),
    type int,
    path varchar(260)
)
```

功能说明：

进行联机备份

参数说明：

db_name：数据库名

name：备份名

desc：备份描述

type：备份类型,0-完全备份，1－增量备份

path：备份存储完整路径

返回值：

1　　备份成功

0　　备份失败，不可知原因

－1　　备份失败，无日志（用于联机备份）
－2　　备份失败，无归档（用于联机备份）
－3　　备份失败，控制文件读取
－4　　备份失败，基础备份信息读取
－5　　备份失败，系统文件信息搜集
－6　　备份失败，备份文件创建
－7　　备份失败，日志文件备份
－8　　备份失败，控制文件备份
－9　　备份失败，备份信息写入
－10　　备份失败，系统处于正在启动，退出或正在进行备份
－11　　备份失败，数据文件备份
－12　　备份失败，备份参数错误
－13　　备份失败，已存在同名备份
－14　　备份失败，已存在同名的备份文件
－15　　备份失败，当前数据库状态不一致

2．SF_BAK_FULL

定义：

```
int sf_bak_full(
    ddb_name varchar(128),
    name varchar(128),
    bdesc varchar(260),
    path varchar(260)
)
```

功能说明：

进行完全联机备份

参数说明：

db_name：数据库名
name：备份名
desc：备份描述
path：备份存储完整路径

返回值：

1　　备份成功
0　　备份失败，不可知原因
－1　　备份失败，无日志（用于联机备份）
－2　　备份失败，无归档（用于联机备份）
－3　　备份失败，控制文件读取
－4　　备份失败，基础备份信息读取
－5　　备份失败，系统文件信息搜集
－6　　备份失败，备份文件创建

－7　　备份失败，日志文件备份
－8　　备份失败，控制文件备份
－9　　备份失败，备份信息写入
－10　　备份失败，系统处于正在启动，退出或正在进行备份
－11　　备份失败，数据文件备份
－12　　备份失败，备份参数错误
－13　　备份失败，已存在同名备份
－14　　备份失败，已存在同名的备份文件
－15　　备份失败，当前数据库状态不一致

3．SF_BAK_INCR

定义：

```
int sf_bak_incr(
      ddb_name varchar(128),
      name varchar(128),
      bdesc varchar(260),
      path varchar(260)
)
```

功能说明：

进行联机增量备份

参数说明：

db_name：数据库名
name：备份名
desc：备份描述
path：备份存储完整路径

返回值：

1　　备份成功
0　　备份失败，不可知原因
－1　　备份失败，无日志（用于联机备份）
－2　　备份失败，无归档（用于联机备份）
－3　　备份失败，控制文件读取
－4　　备份失败，基础备份信息读取
－5　　备份失败，系统文件信息搜集
－6　　备份失败，备份文件创建
－7　　备份失败，日志文件备份
－8　　备份失败，控制文件备份
－9　　备份失败，备份信息写入
－10　　备份失败，系统处于正在启动，退出或正在进行备份
－11　　备份失败，数据文件备份
－12　　备份失败，备份参数错误

－13　　备份失败，已存在同名备份

－14　　备份失败，已存在同名的备份文件

－15　　备份失败，当前数据库状态不一致

4．SF_DEL_BAK

定义：

```
int sf_del_bak(
      name varchar(128),
      db_name varchar(256)
)
```

功能说明：

删除备份

参数说明：

Name：备份名

db_name：数据库名

返回值：

非 0　　操作成功，

0　　操作不成功

5．SF_GET_BAK_BASE_NAME

定义：

```
varchar(256) sf_get_bak_base_name(
      path varchar(256)
)
```

功能说明：

取得一个备份的基础备份名

参数说明：

path：备份的完整路径

返回值：

基础备份名

6．SF_GET_BAK_BY_ID

定义：

```
varchar(256) sf_get_bak_by_id(
      name varchar(128),
      db_name varchar(128)
)
```

功能说明：

按备份名取得服务器端基于某数据库的备份

参数说明：

name：备份名

db_name：数据库名

返回值：

备份的完整路径

7．SF_GET_BAK_BY_NAME

定义：

varchar(256) sf_get_bak_by_name(NAME VARCHAR(128),DB_NAME VARCHAR(128))

功能说明：

取得一个备份的备份路径

参数说明：

NAME：备份名

DB_NAME：数据库名

返回值：

备份路径

8．SF_GET_BAK_DB_ID

定义：

```
int sf_get_bak_db_id(
    path varchar(256)
)
```

功能说明：

取得一个备份的数据库号

参数说明：

Path：备份的完整路径

返回值：

数据库号

9．SF_GET_BAK_DB_NAME

定义：

```
varchar(256) sf_get_bak_db_name(
    path varchar(256)
)
```

功能说明：

取得一个备份的数据库名

参数说明：

path：备份的完整路径

返回值：

数据库名

10．SF_GET_BAK_DESC

定义：

```
varchar(256) sf_get_bak_desc(
    path varchar(256)
)
```

功能说明：

取得一个备份的描述信息

参数说明：

path：备份的完整路径

返回值：

备份描述信息

11．SF_GET_BAK_DIR

定义：

```
varchar(260) sf_get_bak_dir(
      db_id int
)
```

功能说明：

取得备份缺省目录

参数说明：

db_id：数据库 ID

返回值：

当前备份缺省目录

12．SF_GET_BAK_LEVEL

定义：

```
int sf_get_bak_level(
      path varchar(256)
)
```

功能说明：

取得一个备份的备份方式

参数说明：

path：备份的完整路径

返回值：

0　　联机备份

1　　脱机备份

13．SF_GET_BAK_NAME

定义：

```
varchar(256) sf_get_bak_name(
      path varchar(256)
)
```

功能说明：

取得一个备份的备份名

参数说明：

path：备份的完整路径

返回值：

备份名

14．SF_GET_BAK_NUM

定义：

```
int sf_get_bak_num(
    db_name varchar(128)
)
```

功能说明：

取得服务器端基于某数据库的备份的数量

参数说明：

db_name：数据库名

返回值：

备份个数

15．SF_GET_BAK_TIME

定义：

```
varchar(256) sf_get_bak_time(
    path varchar(256)
)
```

功能说明：

取得一个备份的备份时间

参数说明：

path：备份的完整路径

返回值：

字符串形式的备份时间

16．SF_GET_BAK_TYPE

定义：

```
int sf_get_bak_type(
    path varchar(256)
)
```

功能说明：

取得一个备份的备份类型

参数说明：

path：备份的完整路径

返回值：

◆　　完全备份，

1　　增量备份

17．SF_RES_EXEC

定义：

```
int res_exec(void)
```

功能说明：

进行还原，此函数需在 res_prepare 调用之后调用

返回值：

1　　还原成功

0　　还原失败，不可知原因

－1　　还原失败，系统没有找到需要还原的数据库

－2　　还原失败，还原控制文件出错

－3　　还原失败，将主数据库还原为其它数据库，或将其他数据库还原为主数据库

－4　　还原失败，将要被还原的系统与基础备份不一致

－5　　还原失败，取得备份文件列表出错

－6　　还原失败，备份文件打开出错

－7　　还原失败，还原数据文件出错

－8　　还原失败，还原日志文件出错

－9　　还原失败，设置下一个数据库号出错

－10　　还原失败，备份文件有错

18．SF_RES_GET_FILE_NUM

定义：

```
int res_get_file_num(void)
```

功能说明：

取得备份中数据文件的个数，此函数需在 res_prepare 调用之后调用

返回值：

文件个数

19．SF_RES_GET_FILE_PATH

定义：

```
varchar(256) res_get_file_path(
    id int
)
```

功能说明：

按顺序取得备份中数据文件的完整路径，此函数需在 res_prepare 调用之后调用

参数说明：

id：顺序号

返回值：

备份中与顺序号对应的数据文件的完整路径

20．SF_RES_PREPARE

定义：

```
int res_prepare(
    path varchar(250)
)
```

功能说明：

恢复预处理

参数说明：

path：备份路径

返回值：

1　　还原成功

0　　还原失败，不可知原因

－1　　还原失败，系统没有找到需要还原的数据库

－2　　还原失败，还原控制文件出错

－3　　还原失败，将主数据库还原为其它数据库，或将其他数据库还原为主数据库

－4　　还原失败，将要被还原的系统与基础备份不一致

－5　　还原失败，取得备份文件列表出错

－6　　还原失败，备份文件打开出错

－7　　还原失败，还原数据文件出错

－8　　还原失败，还原日志文件出错

－9　　还原失败，设置下一个数据库号出错

－10　　还原失败，备份文件有错

21．SF_RES_RELEASE

定义：

```
int res_release(void)
```

功能说明：

还原子系统释放

返回值：

非 0　　操作成功

0　　操作不成功

22．SF_RES_SET_ARCH_NUM

定义：

```
int res_set_arch_num(
    arch_num int
)
```

功能说明：

设置归档目录的个数，此函数需在 res_prepare 调用之后调用

参数说明：

arch_num：归档目录的个数

返回值：

非 0　　操作成功

0　　操作不成功

23．SF_RES_SET_ARCH_PATH

定义：

```
int res_set_arch_path(
    id int,
    arch_path varchar(256)
)
```

功能说明：

设置归档目录，此函数需在 res_prepare 调用之后调用

参数说明：

id：顺序号，在有多个归档目录需要设置时，从 0 开始将其依次增 1 即可

arch_path：归档目录

返回值：

非 0　　操作成功

0　　　操作不成功

24．SF_RES_SET_FILE_PATH

定义：

```
int res_set_file_path(
      id int,
      file_path varchar(256)
)
```

功能说明：

按顺序设置备份中数据文件的完整路径，此函数需在 res_prepare 调用之后调用，在还原增量备份时此函数不能调用

参数说明：

id：顺序号

file_path：文件路径

返回值：

非 0　　操作成功

0　　　操作不成功

25．SF_SET_BAK_DIR

定义：

```
int sf_set_bak_dir(
      db_id int,
      path varchar(260)
)
```

功能说明：

设置备份缺省目录

参数说明：

db_id：数据库 ID

path：备份缺省目录

返回值：

非 0　　操作成功

0　　　操作不成功

2）日志管理：

1．SF_SET_ARCH_DIR

定义：

int sf_set_arch_dir(
db_id int,
path varchar(260)
)

功能说明：

设置归档缺省目录

参数说明：

db_id：数据库 ID

path：归档缺省目录

返回值：

非 0　　操作成功

0　　　操作不成功

2．SF_SET_ARCH_FLAG

定义：

int sf_set_arch_flag(
db_id int,
int arch_flag
)

功能说明：

设置系统是否归档标志

参数说明：

db_id：数据库 ID

arch_flag：系统是否归档标志，非 0－归档，0－不归档

返回值：

非 0　　操作成功

0　　　操作不成功

3．SF_SET_ARCH_STYLE

定义：

int sf_set_arch_style(
db_id int,
style varchar(128)
)

功能说明：

设置归档文件产生方式

参数说明：

db_id: 数据库 ID

style: 归档文件产生方式

返回值：

非 0　　操作成功，

0　　　操作不成功

4．SF_SET_LOG_PATH

定义：

```
int sf_set_log_path(
      db_id int,
      path varchar(260)
)
```

功能说明：

设置日志文件完整路径，系统需重新启动才生效

参数说明：

db_id：数据库 ID

path：日志文件完整路径

返回值：

非 0　　操作成功

0　　　操作不成功

5．SF_SET_MAX_LOG_LEN

定义：

```
int sf_set_max_log_len(
      db_id int,
      len int
)
```

功能说明：

设置日志文件最大长度

参数说明：

db_id：数据库 ID

len：日志文件最大长度，以兆为单位

返回值：

非 0　　操作成功

0　　　操作不成功

6．SF_CKPT

定义：

```
int sf_ckpt(
      arch_flag int
)
```

功能说明：

手工设置检查点

参数说明：

arch_flag：

非 0　　产生归档检查点

0　　　产生一般检查点

返回值：

无意义

7．SF_GET_ARCH_DIR

定义：

```
varchar(260) sf_get_arch_dir(
    db_id int
)
```

功能说明：

取得归档缺省目录

参数说明：

db_id：数据库 ID

返回值：

当前归档缺省目录

8．SF_GET_ARCH_FLAG

定义：

```
int sf_get_arch_flag(
    db_id int
)
```

功能说明：

取得系统是否归档标志

参数说明：

db_id：数据库 ID

返回值：

非 0　　归档

0　　　不归档

9．SF_GET_ARCH_STYLE

定义：

```
varchar(128) sf_get_arch_style(
    db_id int
)
```

功能说明：

取得归档文件产生方式

参数说明：

db_id：数据库 ID

返回值：

归档文件产生方式

10．SF_GET_LOG_PATH

定义：

```
varchar(260) sf_get_log_path(
    db_id int
```

)

功能说明：

取得日志文件完整路径

参数说明：

db_id: 数据库 ID

返回值：

日志文件完整路径

11．SF_GET_MAX_LOG_LEN

定义：

int sf_get_max_log_len(db_id int);

功能说明：

取得日志文件最大长度

参数说明：

db_id：数据库 ID

返回值：

日志文件最大长度

附录 5　DM 技术支持

如果您在安装或使用 DM 数据库及其相应产品时出现了问题，请首先访问我们的 Web 站点 http://www.dameng.cn。在此站点我们收集整理了安装使用过程中一些常见问题的解决办法，相信会对您有所帮助。

您也可以通过以下途径与我们联系，我们的技术支持工程师会为您提供服务。

技术支持：technology@dameng.cn

客户支持：support@dameng.cn

电话：

产品销售：027-87522500 转 8016

客户支持：027-87522500 转 8035

通讯地址：武汉市洪山区珞瑜路 243 号华工科技产业大厦 9 楼

武汉华工达梦数据库有限公司

邮政编码：430074